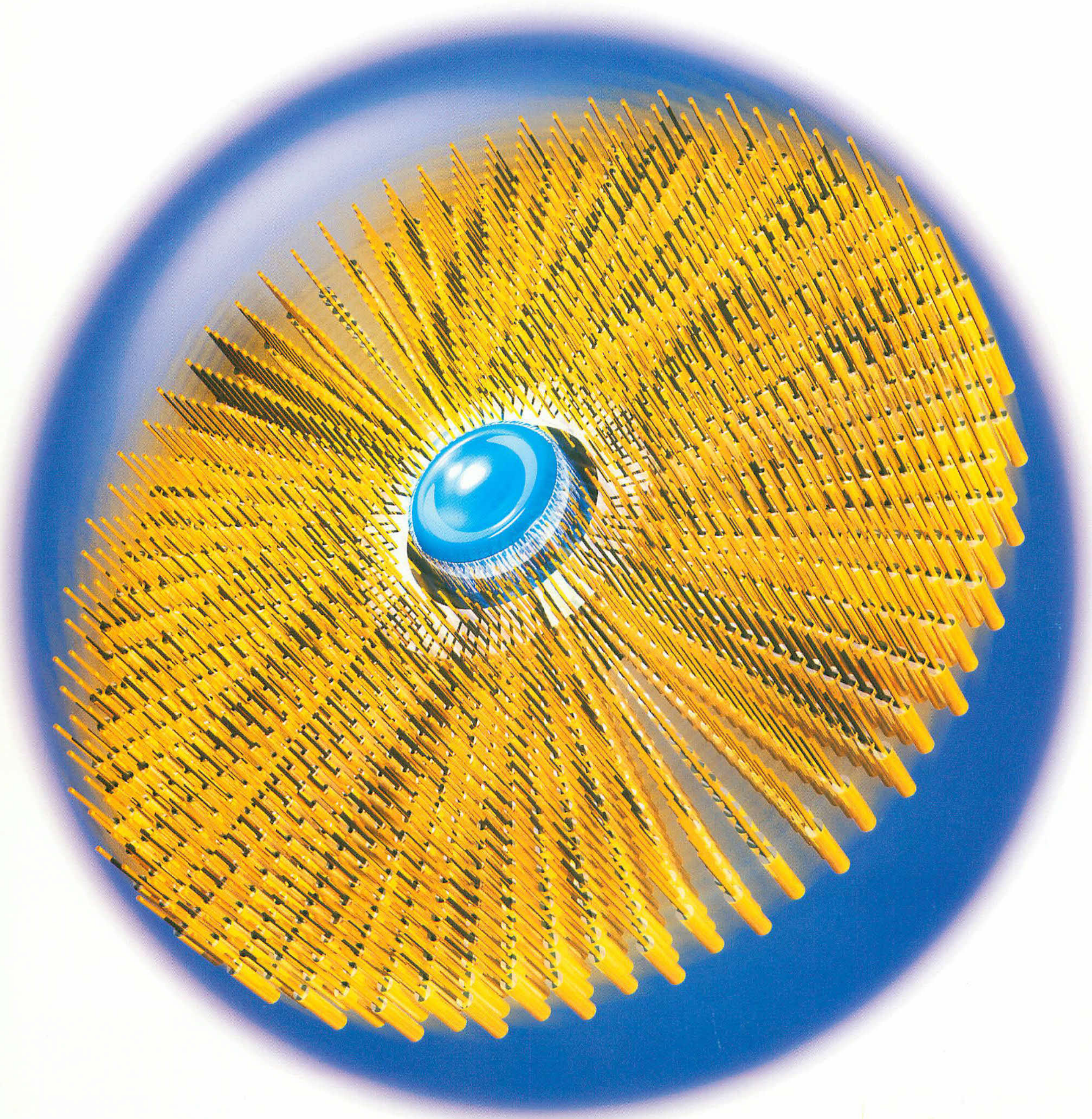
インターフェース増刊

マイクロプロセッサ・アーキテクチャ入門

RISCプロセッサの基礎から最新プロセッサのしくみまで

中森章 著

TECH I
Technology Interface



Vol.20

CQ出版社

TECH I シリーズ 　　**好評発売中**



Vol.19 OSの移植からGUIによるアプリケーション開発まで

実践リアルタイムOS活用技法

B5判 152ページ
定価2,000円

組み込み機器の開発に欠かせない存在としてリアルタイムOSが注目を集めている．リアルタイムOSは，制限時間内の応答が要求されるだけでなく，組み込み機器での使用のために高速な起動，限られた資源で動作することが要求されるなど，デスクトップOSとは違った知識が要求される．本書は，組み込み機器でリアルタイムOSを使い，GUIアプリケーションを作成するために必要となる知識について詳しく解説する．



Vol.18 ARMアーキテクチャの詳細＆ARM7/XScaleの応用

ARMプロセッサ入門

B5判　208ページ
CD-ROM付き
定価2,200円
ISBN4-7898-3329-1

本書は，ARMプロセッサファミリの基礎知識からアーキテクチャの詳細，アセンブラ命令や最適化について，またコンパイラやデバッガ，開発環境など，ARMプロセッサ全般に関して解説する．さらに，実際に外販されているプロセッサを搭載した評価ボードなどを取り上げ，その上で動作する実際のハードウェア応用例，プログラミング事例などを解説する．



Vol.17 実践μITRONプログラミング

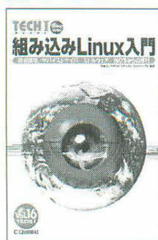
リアルタイムOSと組み込み技術の基礎

B5判
200ページ
定価2,200円
ISBN4-7898-3328-3

高田 広章 監修・著　岸田 昌巳／宿口 雅弘／南角 茂樹 著

組み込みシステムとは，いろいろな機械や機器に組み込まれてその制御を行うコンピュータシステムのことであり，最近は適用分野が急速に広がっている．また，技術の進歩に合わせてソフトウェアの大規模化・複雑化が進んでおり，リアルタイムOSを用いることが不可欠となっている．

本書は，汎用オペレーティングシステムに関する一般的な知識と併せて，μITRONを例としたリアルタイムOSの活用技法について解説した．



Vol.16 開発環境/デバイスドライバ/ミドルウェア/他OSからの移行

組み込みLinux入門

日本エンベデッド リナックス コンソーシアム 監修
B5判　272ページ　定価2,200円
ISBN4-7898-3327-5

本書では，組み込みシステムの開発にLinuxを使うための技術要素を，入門者向けに，総覧的に解説している．内容としては，組み込みLinuxの現状，開発環境，カーネル/デバイスドライバ，ミドルウェア，他OSからの移行などを盛り込んでいる．また，組み込みLinuxに関連するキーマンへのインタビューも収録している．



Vol.15 組み込みシステムの基本とタスクスケジューリング技術の基礎

リアルタイム/マルチタスクシステムの徹底研究

藤倉 俊幸 著
B5判
264ページ
定価2,200円
ISBN4-7898-3326-7

携帯電話，デジカメ，冷蔵庫，洗濯機，湯沸かしポット，PDA，ガスメータ…….これらはマイコンが組み込まれ，さまざまな制御を行う組み込み機器である．本書では，これら組み込み機器を開発するときのキーワードとなる，マルチタスク，リアルタイム，テスト，状態マシン，プライオリティインヘリタンスなどを一つ一つとりあげ，ていねいに解説している．



Vol.14 規格の概要からカード/ホストコントローラ/ドライバの設計/製作

PCカード/メモリカードの徹底研究

B5判　280ページ　CD-ROM付き　定価2,200円　ISBN4-7898-3325-9

本書では，さまざまなメモリカードの規格や仕組みを詳しく解説する．またユーザー独自仕様のPCカードのハードウェアとドライバを開発する方法について，さらには組み込み機器でメモリカードを使うためのホストインターフェースの設計事例も解説する．

CQ出版社 〒170-8461 東京都豊島区巣鴨1-14-2 販売部 ☎03-5395-2141 振替00100-7-10665

定価は2004年3月現在の消費税込みの価格です

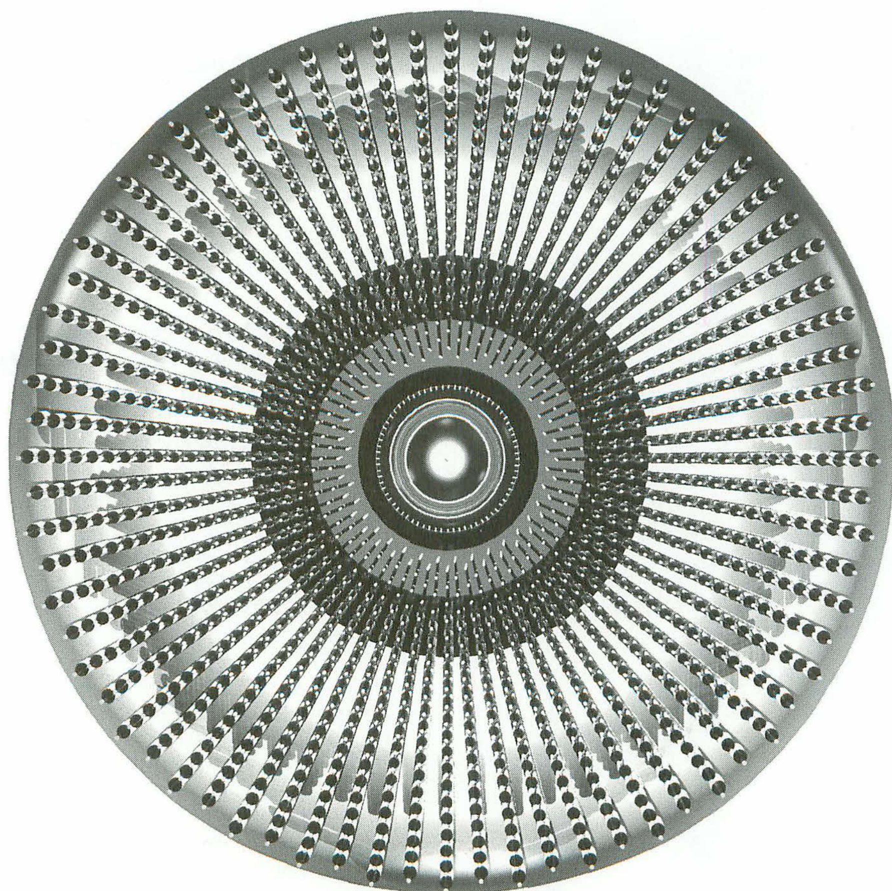
CQ出版社のURL　http://www.cqpub.co.jp/

マイクロプロセッサ・アーキテクチャ入門

RISCプロセッサの基礎から最新プロセッサのしくみまで

中森章 著

TECH I
Technology Interface



CQ出版社

目次

CONTENTS

表紙デザイン　MINO MIED・K
本文デザイン/レイアウト　(有)みやこワードシステム

Prologue

コンピュータの誕生からプロセッサの発展まで

マイクロプロセッサの歴史

そもそもマイクロプロセッサ(MPU)とは何なのだろうか．多くの人は，MPUがMicro Processing Unit(小型処理装置)の略語であり，コンピュータの中心的な動作を制御するLSIであることを知っている．その意味でCPU(Central Processing Unit：中央処理装置)と呼ばれることもある．本書では一部を除きMPUと呼ぼう．

では，なぜMPUに8ビット，16ビットあるいは32ビットという種類があるのか，なぜ16ビットMPUよりも32ビットMPUのほうが処理性能が優れているのか，という点について知っている人は意外と少ないのではないだろうか．

これを明らかにするには，大型計算機の歴史から振り返ってみる必要がある．歴史を探ることで，MPUの未来もおのずと見えてくるのではないだろうか．

● コンピュータ(計算機)という発想はいつから？

計算の機械化は古くから考案されてきた．16世紀にフランスの数学者のパスカル(Blaise Pascal)が考案したパスカリーヌ(Pascaline)をはじめとして，ドイツの数学者であるライプニッツ(Gottfried Wilhelm Leibniz)がパスカリーヌを拡張した計算機を完成させている．

現在のイメージに近いコンピュータを最初に構想したのは，19世紀のイギリスの数学者であるバベッジ(Charles Babbage)といわれている．彼は1819年頃から，高信頼度の数表を階差法により作成する歯車式の階差機関の作成に着手した．1832年には試作機が作成されたが，政治的な理由で階差機関の開発は成功しなかった．

その直後，階差機関の計算能力を上げる目的で，バベッジは解析機関を発案した．その複雑なメカニズムを実現するために，データ(記憶領域)と演算を分離することが考えられた．データから独立した演算器は，一連の指令(プログラム)を与えることで，種々の計算に対応できた．これがプログラム制御のはしりである．

階差機関は，階差法という計算に特定された専用マシンであったが，解析機関はある程度の汎用性をもっていた．これをもって，解析機関を最初のコンピュータとみなす向きもあるが，プログラム内蔵方式ではなかったし，条件分岐機構をもっていないという意味では，コンピュータではないという意見もある．

その後，バベッジの業績は何人かの研究者に受け継がれたが，どれも試作程度で終わっている．バベッジ以後，1940年代までコンピュータ開発の表立った動きはなかったというのが定説である．この期間は100年の空白といわれている．

もっとも，これは米国を中心とした史観であり，実際には1930年代に機械式やリレー式の計算機がドイツのツーゼ(Konrad Zuse)らによって研究/試作され，またシュレイヤー(Schreyer)やアタナソフ(John V. Atanasoff)らが真空管方式によって開発を始めている．ツーゼのコンピュータは「Z1」，アタナソフのコンピュータは「ABC」として歴史に名を残している．

チューリング(Alan Turing)は1936年にチューリング・マシンに関する論文を発表し，これが現代コンピュータの基礎理論となっている．チューリングもまた，第二次世界大戦中に暗号を解読するための「ボンベ」というチューリング・マシンを応用した機械(コンピュータの原点)を開発している．ボンベはリレーを使用していたが，真空管を使用した電子計算機もチューリングの提案で開発された．これも暗号解読用である．

リレーは電磁石でスイッチをON/OFFするものだが，電気で同様の処理を行う真空管を使用すると1,000倍近い計算速度を得ることができる．これは1943年

にCOLOSSUSという計算機として実現している．

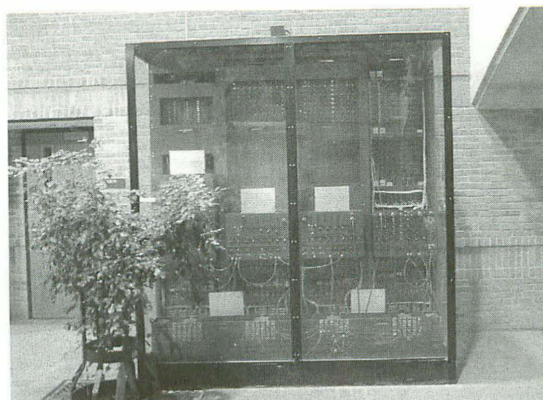
これはイギリスの話であったが，アメリカでも同時期に真空管を使用したENIAC(Electronic Numerical Integrator and Computer)という計算機が開発されていた．ENIACは実戦には間に合わなかったが，COLOSSUSは戦時中稼働していた唯一のコンピュータとして，後世に名を残している．それは，戦後も30年にわたって諜報活動に活用されるが，当時は機密事項として関係者が知るのみであった．

● アタナソフのコンピュータ──ABCマシン

1970年頃までの定説では，世界最初のコンピュータはモークリー(John W. Mauchly)とエッカート(J. Presper Eckert)およびゴールドスタイン(Herman H. Goldstine)によって1945年に開発されたENIACということになっていた．その説に一石を投じたのが，1937年からアタナソフとベリー(Clifford E. Berry)の開発したABC(Atanasoff-Berry-Computer)マシンである．これは，ガウスの消去法を想定した真空管式の計算機で，1942年にはほとんど完成していたといわれる．現在では，このABCマシンこそがENIACに先立つ世界最初のコンピュータといわれることが多い．

もともとアタナソフの業績は世間から忘れ去られていた．それを白日の下に引き戻したのは，1960年代に始まった，ENIACの基本特許に関する係争である．この裁判でENIACの特許が無効になったが，その根拠の一つとして，モークリーがアタナソフからENIACの基本原理を得ていたということが挙げられた．

事実，1941年にモークリーはアタナソフを訪問してABCマシンを見学していた．かくしてアタナソフの名前は一躍クローズアップされることになる．しかし，ABCマシンは29変数までの連立一次方程式の解法機にすぎず，プログラム内蔵という観点から見てもコンピュータと呼ぶにはふさわしくない．



写真A　ミシガン大学に展示されているENIAC
(写真提供：近藤和彦氏)

● 真空管のコンピュータ

初期のコンピュータは，人間が手計算でやっていてはとても終了しないほど多量の計算を高速に行わせるために開発された．昔はそれほどの需要があったわけではないが，第二次世界大戦の頃になると大規模な計算の必要性が顕在化してきた．その主たる用途が軍事目的であったことは否めない．

現在のコンピュータのはしりは1945年にペンシルバニア大学で作られたENIACといわれているが，これは大砲の弾道計算をするために作られたコンピュータである．その処理能力は現在のコンピュータと比べてはかわいそうなくらい低く，どちらかというとプログラム電卓といった感が強かったようだ．

ENIACは，ある程度のプログラムを内蔵することもできたし，条件分岐機構ももっていたので，最初のコンピュータという栄誉を受ける資格は十分にある．

Column　バグの起源

コンピュータのソフトウェアやハードウェアの誤り(不具合)をバグ(虫)というが，この起源をENIACに求める説がある．蛾がコンピュータの真空管の間に紛れ込んで誤動作を誘発させたというのだ．

これは昔よく言われたエピソードであるが，現実は少し事情が異なる．正式には，COBOLの開発者であり世界でもっとも有名なプログラマであるグレース・ホッパー(Grace Murray Hopper)が，初めてバグという言葉を使ったとされている．

彼女がハーバード大学で計算機(Mark-II)の開発に携わっていたとき，電子回路に蛾が迷い込んで故障したのが発端である．その蛾は記録簿に貼り付けられて保存されており，現在はワシントンD.C.のスミソニアン博物館で見ることができるという．

ただし，これを主張する学者はCOLOSSUSの存在を無視しているようにも思える．ただ，COLOSSUSは暗号解読専用という観点から，汎用コンピュータとは認めてもらえないのだ．

1989年，米国のスミソニアン協会がアメリカ歴史博物館でコンピュータ開発の歴史展示を試みたとき，コンピュータの発明者はモークリーとエッカートになっていた．それが政治的な圧力でうやむやな表現に変更された．その中で，アタナソフは最初のコンピュータを発明したが動作させることはできなかった，と説明されたという．

ENIACの本体は，30m × 90m × 3mの筐体の中に18,000本の真空管と10,000個のコンデンサを詰め込んだものである．このため，ENIACを設置するためにはまるまる一部屋分のスペースが必要だった．また，多くの真空管を動作させるために機関車並みの電力が必要だったという．

それでも，電気式の真空管を使用するため，計算機の処理能力は機械式のリレーに比べると飛躍的に向上した．しかし，真空管を使っているために「図体がでかい」,「熱い」,「壊れやすい」というのが当時のコンピュータの常識だったようだ．この真空管の問題を何とかしない限り，コンピュータの発展はありえなかった．

● フォン・ノイマンに対する誤解

フォン・ノイマン(Jhon von Neumann)は，今日のコンピュータアーキテクチャの基礎を創造した人物として広く知れ渡っている．事実，プログラム内蔵を基本とする今日のコンピュータは「ノイマン型」といわれている．しかし，これはモークリーやエッカートの名誉を著しく傷つけるものである．

1944年の初め，ENIACの設計が始まってから18か月が過ぎた頃，フォン・ノイマンはゴールドスタインと会う機会を得た．そこで，フォン・ノイマンはゴールドスタインが関わっていた現在進行中のENIACの計画に非常に興味を覚えた．当時フォン・ノイマンは，原子爆弾を開発するマンハッタン計画の顧問をしていたが，この戦争に役立ちそうなコンピュータのことは知らされていなかった．これは，ENIACのスポンサーともいえる国防研究委員会(NDRC：National Defense Research Committee)がENIACを信用しておらず,取るに足らないものと考えていたからである．しかし，フォン・ノイマンは非常に興味を覚え，1944年9月にENIACの開発現場を訪れ，ENIACの秘密情報へのアクセスが許された．

さて，ENIACにはプログラミングが難しい，メモリが少量しかないという欠点があった．関係者の多くはENIACが完成するかなり前から，後継機種のEDVAC(Electronic Discrete Variable Automatic Computer)の議論を始めている．そして，1945年3月，フォン・ノイマン，モークリー，エッカート，ゴールドスタインらがEDVACの設計に関して議論した記録が，いわゆる「EDVACレポート」として，フォン・ノイマンの単独名で，機密事項であるにもかかわらず，世界のコンピュータ技術者の間に広く流布された．

フォン・ノイマンがEDVACの設計に参加する前にプログラム内蔵方式は考案されていた．しかし，この文書により，フォン・ノイマンがプログラム内蔵方式のコンピュータの発明者として誤って伝わってしまったのである．フォン・ノイマンは発明者ではないが，プログラム内蔵方式を論理的に明確にして発展させた業績は認めるべきであろう．

EDVACは，関係者間の意見の対立により大幅に開発が遅れ，ついに頓挫してしまう．一方，フォン・ノイマンはプリンストン大学で新しいコンピュータの開発を指導することになる．そんな中，世界最初のプログラム内蔵方式のコンピュータとしての栄誉を勝ち取ったのは，EDVACの影響下にイギリスのケンブリッジ大学でウィルクス(Maulice Wilkes)により製作され，1949年5月に稼働を始めたEDSAC(Electronic Delay Storage Automatic Calculator)である．この名称はEDVACを意識して付けられたという．なお，ウィルクスはマイクロプログラミングの提唱者としても有名である．

さて，これ以後も計算機の試作は星の数ほど行われ，IBMなどの大型計算機やスーパーコンピュータの開発へとつながっていくのだが，その歴史を追うことは筆者の意図ではない．今後はマイクロプロセッサの進歩を主に見ていこう．

● トランジスタが登場した！

コンピュータにとっての朗報は，1947年も終わりに近付いたクリスマスの2日前に訪れた．ベル研究所のウィリアム・ショックリー(William Shockley)，ジョン・バーディーン(John Bardeen)，ウォルター・ブラッテン(Walter Brattain)によってトランジスタが発明されたのだ．

言うまでもなくトランジスタは，半導体産業において20世紀最大の発明である．トランジスタは真空管と違って熱をもたないし，壊れにくく，真空管より高速に動作する．そして，サイズが小さいのがなによりの利点だった．このトランジスタは，ラジオや補聴器など多くの電子機器の中心的デバイスとして確固たる地位を築いていくことになる．

当然，トランジスタを用いたコンピュータも作られた．FORTRANやCOBOLといった高級言語のコンパイラが登場したのは，トランジスタのコンピュータが全盛になる1950年代の後半から1960年代にかけてのことだった．この時期の代表的なコンピュータとして，IBMの7070や7090がある．まだ，マイクロプロセッサは誕生していない．

さて，トランジスタを用いてコンピュータを作る場合，最大の問題点は回路規模が大きく複雑であるということだった．数百ものトランジスタやコンデンサをはんだ付けしていく作業は人間の手によっていたが，それでいて十分な信頼性を得るのは至難の技だった．

その障壁を乗り越えてコンピュータを作ったのだから，当時のコンピュータメーカーの頑張りには脱帽する．しかし，力まかせに作るコンピュータにはおのずと限界がある．人類には理論的には可能であっても，実装技術の未熟さゆえに到達できない夢がいくつもあったのだ．

● 集積化の時代

コンピュータにとって第2の転機は，1959年に訪れた．Texas Insturuments(TI)のジャック・キルビー(Jack Killbey)とIntelの創始者の一人であるロバート・ノイス(Robert Noyce)が，シリコンウエハ上に抵抗やコンデンサを作るというアイデアを実現させたからだ．これがIC(集積回路)の誕生である．

それまでは構成要素が独立した多くの部品であったため，それらを接続する困難さが生じる．一つのチップに構成要素を作り込んでやれば接続の手間が省けるばかりでなく，非常に小型化できるというのがその基本的なアイデアである．もちろん，思い付きだけでICを製造できるわけではないが，とにかく，数々の製造上の困難を乗り越えた奇跡のチップとしてICが誕生した．

そして，ICが登場してから10年後の1969年には，月面上に小さいけれど人類にとっては大きな一歩を印すことになった．あのアポロ計画である．しかし，合理的なアメリカ人が道楽(?!)で月まで行くわけはない．その背後に，宇宙の軍事利用という暗い影を宿していたことは厳然たる事実である．

しかし，その技術が民間用に転換されてきて，われわれ一般人もその恩恵に浴すことができたのは一筋の光明かもしれない．ICが初めて応用されたのは補聴器だったというし，ICがなければ現在のように信頼性の高いテレビ，ビデオ，DVDプレーヤなどのAV機器を手にすることもなかったはずである．

● マイクロプロセッサの鼓動

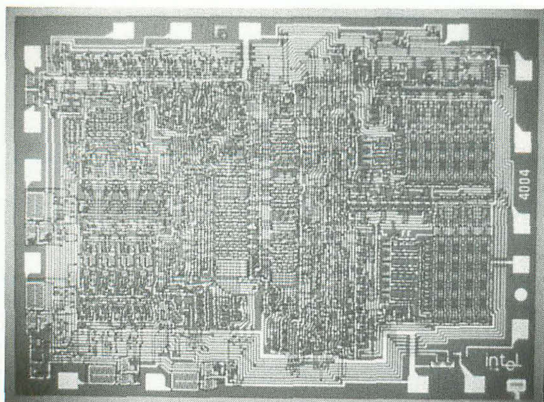
マイクロプロセッサの誕生には，日本が大きく関わっている．なぜなら，マイクロプロセッサの物語は東京に端を発するからである．

1968年，日本の事務機器メーカーであるビジコン社は画期的なプリンタ付き電卓を作った．この電卓はプログラムをROMから読み出して実行するという，現在のコンピュータに近い形式を採用していた．しかもこの電卓は，ROMの内容を変更するだけでまったく別の電卓を作ることができるようになっていた．

1969年に入ると，電卓の高性能化，多様化，低価格化，高信頼化などの要請から，電卓をLSI化する計画が生まれた．しかし悲しいことに，日本には電卓程度の複雑さをもつ回路をLSI化する技術すらなかった．そこでビジコン社は，Intelに援助を求めた．そのときIntelは，ビジコン社側の示すLSIの規模が他社の電卓用LSIに比べて大きい(約16個の異なるチップを使用する)ので商売にならないと判断し，代わりに4ビットのMPUというアイデアを提示してきた(本音はチップ1個分の開発コストしかなかった)．

これは，電卓のプログラムに使われていた命令をもっと低レベルの機械語レベルに引き下げて，汎用性をもったLSIをねらったものである．ビジコン社はもともとプログラム方式の電卓を作っていた経験から，この新しいアイデアをすんなりと受け入れることができた．結果として，ビジコン社とIntelの折衷案で，世界初の4ビットMPUが作られることになったという．

電卓用のLSIに見られるように，当時のLSIの多くはカスタムデザイン(固有の目的のための設計)によって作られていた．特殊な目的をもったLSIを数多く短期間に製造していくためには，プログラム可能な汎用LSIというアプローチは非常に有用な解答だった．これが，ゆくゆくは「部品としてのコンピュータ」という市場を産むことになる．



写真B　初のマイクロプロセッサ4004（写真提供：嶋正利氏）

今ではIntelの主要製品となっているMPUであるが，当初Intelの幹部はその将来性に気付いていなかった．電卓の部品という認識で，メモリ事業が主体のIntelにとっては，サイドビジネスの一環でしかなかった．MPUがIntelの未来を切り開く存在であることに気付くのは，発表から15年も経った後だという．

Intelが4004の所有権をもつようにしたのは，積極的な理由からではなかった．4004の開発中に電卓事業が不振になり，ビジコン社側が契約料の大幅値下げを要求してきたためだ．Intelは，電卓市場以外においても4004の外販権を無料で提供することを条件に，その要求を呑んだ．

かくして，1971年11月15日，Intelは4004を世界初のマイクロプロセッサとして発表した．これがマイクロプロセッサの誕生である．当時のIntelの最高経営責任者は，「人類史上でもっとも革命的な製品の一つ」とコメントしたらしい．しかし，多くの人々は4004のコンセプトを理解できず，Intelを脅威と思う人はほとんどいなかった．革新的な製品の最初は，おしなべてこんなものかもしれない．

4004は750kHzのクロックで動作し，1命令の実行には最低8クロックが必要だった．これは，1クロック実行が当然という現在のRISC技術から見れば隔世の感がある．4004のクロックに関しては，Intelの公式資料では108kHzとなっているが，これは誤りである．4004のニュース・リリースで，命令の実行速度が10.8μsとなっていたのを勘違いしたものと思われる．この後に登場する8008の動作周波数も200kHzとなっている文献が多いが，500kHzの誤りである．

● マイクロプロセッサの展開

1972年4月1日，Intelは4004のアーキテクチャを拡張して8ビットデータ（文字データ）を扱えるようにした8ビットMPUの8008を発表した．動作周波数は500kHzだった（後に800kHz品も開発される）．8008は，テキサスの端末メーカーであるデータポイント社からの受注である．しかし，データポイント社が契約料を払えなかったため，Intelは8008の命令セットの使用権とチップの外販権を獲得する．のちのx86命令セットの萌芽である．

8008は4004の後継と説明されることが多いが，実は4004の開発中にその技術者を引き抜いて開発したという．その意味では，4004と同世代の兄弟チップである．

そして1974年，8008の改良版である，同じ8ビットMPUの8080が発表されるにあたって，マイクロプロセッサが本格的に市場に受け入れられるようになった．そして人々は，マイクロプロセッサによって多くの製品に知能を与えることができると考え，無数の新しい応用を夢に描いていった．

約55年前，部屋いっぱいの設置場所と機関車並みの電力を必要としたコンピュータが小指大（現在の規模では親指大というほうが適切か）のマイクロプロセッサへと凝縮されることで，コンピュータは日常生活の基本的な枠組みの中へ浸透していくようになった．

とはいえ，マイクロプロセッサの誕生が電卓用LSIをきっかけとしたように，初期におけるマイクロプロセッサの役割は，既存の制御機器の置き換えが主目的だった．この場合，とにかく動くことが第一で，プログラムの生産性や性能は二の次だった．

小型で動けばよいという時代を過ぎると，当然のことながら，マイクロプロセッサは性能を要求されることになる．そこで，マイクロプロセッサは8，16，32とビット数を増やしながら大型計算機の進歩を大急ぎで追いかけていった．

そして，現在の32ビットマイクロプロセッサの処理能力は大型計算機の処理能力に近づき（ある意味では凌駕し），コンピュータごとに専用のMPUを使用していたミニコンのMPUすら駆逐してしまった感がある．

また，その応用分野もエンジニアリングワークステーション，画像処理システム，音声処理システム，ロボット制御，プロセス制御，人工知能システムなど，多種多様の分野に広がるようになった．

●「もし」の世界……世界最初のMPUは日本製だったかもしれない

上述したとおり，MPUの歴史は，1971年のIntelの4004の発売に始まったといわれている．しかし，4004に遅れること数か月，日本でもμPD700という2チップ構成のMPUが開発されていることはあまり知られていない．

これは，シャープがコカ・コーラ社から依頼を受けて論理設計を行い，NECが製造した4ビットのプロセッサである．シャープは，最初，三菱電機に製造を依頼したのだが，これが見事に失敗してしまう．そして，NECにお鉢が回ってきて日の目を見たわけだ．もし，シャープが初めからNECに依頼をしていれば，世界最初のMPUは日本製ということになっていたかもしれない．

その後，NECがμPD700の権利を買い取り，1チップのμCOM4として発売された．このMPUはキャッシュレジスタを中心に広く応用されたという．どちらかといえばIntelよりもMotorolaのMPUに近い命令セットで，使い勝手が良かったらしい．

4ビットMPUは，その後の低価格化の要求から，1チップマイコンが市場の中心になっていき，家電製品に採用されるようになる．この分野は日本の独壇上である．1チップマイコンは4ビット，8ビットと独自の進化を遂げていくが，16ビット以降になると米国製のMPUに置き換えられていく運命をたどった．

日本の半導体史における国産第一号のマイコンは，1973年に発表された東芝製のTLCS-12ということになっている場合もある．これは最初から12ビットプロセッサであったことが画期的である．Intelでは8ビットの8008が発表されたばかりである．

TLCS-12は米国フォード社の思惑が絡んで開発された．1970年に米国では自動車の排ガスを規制するマスキー法が成立し，自動車メーカーはその対応を迫られていた．東芝とフォードは，協力してエンジン制御の自動化をマイクロプロセッサに託したのだ．

● 32ビットMPUの条件

MPUにおける8ビット，16ビット，32ビットというビット数の増大は，扱うプログラムの規模が大きくなったことを意味する．プログラムが複雑になるにつれて，コードサイズは大きくなる．また，大量のデータを扱うためには，それぞれのデータにアドレス付けができなければならない．バイトアドレスを採用する場合，アドレス空間の大きさはアドレス長が4ビットで16バイト，8ビットで256バイト，16ビットで64Kバイト，32ビットで4Gバイトである．

20年前は，32ビット長より大きいアドレスを扱うようなマイクロプロセッサは登場していない．さすがに，MPUのアーキテクチャ設計者たちは，アドレス空間は4Gバイトもあれば十分だと思ったのだろう．32ビットMPUであるx86系のアドレス空間は64Tバイト(48ビット)と言われているが，セグメント切り替えが入るので実質は4Gバイトである．

アドレス空間は4Gバイトもあれば十分だが，初期の大型計算機は経済的な理由(メモリが高価だった)からMPUのアドレスバスを32本用意することはナンセンスだった．また，MPUのピン数が増えると周辺回路が複雑になるので，MPUチップから出ているアドレスやデータのピン数は必要最少限に抑えられていた．そして，制限されたアドレス空間を有効に使うための技術として仮想記憶という方法が考案された．

どんなに大規模なプログラムであっても，プログラムのすべての部分を同時に実行することはない．動的に眺めれば，プログラムは主記憶上のある小さな領域でしか実行されていない．データも瞬間瞬間に扱う量はわずかである．

仮想記憶は，このようなプログラムの局所性を考慮し，プログラムのうち，現在実行していない部分は，ハードディスクなどの2次記憶にしまっておこうとするアイデアである．ディスク上のプログラムは必要があれば主記憶にロードし，その代わり，前に主記憶にあったプログラムをディスクに退避する．

そして，この仮想記憶を行う場合に必要になるのがアドレス変換である．アドレス変換は，プログラムの中で使われているアドレス(論理アドレス，または仮想アドレスと言う)を実際の主記憶に収まる範囲のアドレス(物理アドレス，または実アドレスと言う)に見せかけるメカニズムである．

そして，このアドレス変換を行うデバイスがMMU(Memory Management Unit)である．32ビット以前のMPUでは，MPUに外付けするMMUを使う場合が多かったが，最近の32ビットMPUではMMUを内蔵するのが常識になってきている．

● マルチタスクと仮想記憶

論理アドレスが32ビット，物理アドレスが32ビットあるMPUに，なぜ仮想記憶が必要か疑問に思う人

がいるかもしれない．実は，仮想記憶には少ない記憶容量を大きくみせかけて使用するというほかにも大きな意義がある．

すなわち，マルチタスクを行う場合，各タスク（プログラム）ごとに4Gバイトのアドレス空間を提供するという役割を果たす．マルチタスクの環境下では，主記憶上には多くのタスクが混在して置かれている．主記憶上にあるプログラムしか実行できないというのは，フォン・ノイマン型MPUの宿命である．すべてのプログラムは，たとえば，0番地（論理アドレス）から開始されているが，そのプログラムが実際にロードされている主記憶は0番地（物理アドレス）とは限らない．アドレス変換によって，それぞれのタスクが置かれている主記憶領域を0番地から始まっているようにみせかけているために，すべてがうまくいっているように見える．

ここで，プログラムを主記憶上のどの位置に持ってきても動作させることができる，ポジションインディペンデントという言葉がある．これは，規模が小さく，MMUを内蔵していないMPUにおいてマルチタスクを行わせるための苦肉の策とみることもできる．また，セグメント方式もセグメントのベースアドレスに対してはポジションインディペンデントであり，これも広義の仮想記憶といえる．

とにかく，マルチタスクを行う場合，それぞれのタスクごとに主記憶とディスクの入れ替え（スワッピングという）を行ってやれば，タスクごとに4Gバイトのアドレス空間を割り当てることができるというわけである．

ここで，スワッピングや論理アドレスと物理アドレスを具体的に対応付けるのはOSの役目である．この意味で，32ビットMPUはOSの介在を強要する…というか，OSが必須なMPUである．

● 32ビットMPUに要求されるアーキテクチャ

このように眺めてくると，32ビットMPUに要求される基本的なアーキテクチャが浮き上がってくる．それは次の三つである．

(1) 大規模なプログラムに対応できるように，論理アドレスとして32ビットを提供すること（レジスタはポインタとして使用するので32ビットでなければならない）

(2) 大規模なプログラムは高級言語で記述されることが多いので，高級言語のサポートを容易にすること

(3) 仮想記憶が常識になるので，OSのサポートを容易にすること．

これらは，現在主流のRISCのアーキテクチャとは相容れない部分があるかもしれないが，RISC以前のMPUはこれらに基づいていたことは間違いない．

これらにさらに付け加えるならば，MPUの汎用化が進んでくると，従来の整数演算に加えて浮動小数点演算が行えることも必要になってくる．また，一つのMPUだけで処理を行うのでなく，複数のMPUが協力して処理を行うことで処理性能をあげるマルチプロセッサへの対応も必要になってくる．

今後，32ビットMPUを設計する場合，これらの機能のサポートも重要になってくるに違いない．(2)に関しては，人によっては異論があるだろう．コンパイラの性能がよければ，MPUの命令セットはどうでもよいという考え方である．これは，CISCとRISCの優位性の議論に密接に関係する．

● 32ビットMPUのハードウェアの特徴

以上のようなアーキテクチャの拡張から，自然発生してくる特徴を有しているだけでなく，32ビットMPUでは個々の命令の処理速度を上げるために，従来は大型計算機で使われてきたハードウェア技術を採用していることも大きな特徴である．

8ビットや16ビットMPUの頃は，実装技術の制限によって実現できなかった機能が，32ビットMPUではどんどん取り入れられている．これらの技術は，具体的にはパイプライン制御とキャッシュメモリの内蔵である．

(1) パイプライン制御

MPUの命令の処理を大雑把にみると，「命令フェッチ」，「命令デコード」，「実行」，そしてその命令の動作により「ライトバック」などの処理が発生する．従来の8ビットや16ビットMPUでは，これらの処理を直列的に行っていたので，一つの命令を実行するために三つあるいは四つ以上の段階を経なければならなかった．しかし，ある命令を実行しているときに，次の命令のデコードをすることは可能である．また，ある命令をデコードしているときに，次の命令をフェッチすることも可能である．このように「フェッチ」，「デコード」，「実行」という命令の処理の段階は並列に実行することが可能なのだ．この命令の並列処理をパイプライン制御という．

パイプライン制御を行えば，ある命令のフェッチやデコードは他の命令の実行時間に隠れてしまうので，命令のフェッチとデコードの時間を0にすることができるというわけである．

パイプラインの段階が，「フェッチ」，「デコード」，「実行」の三つに分かれているとき，それを3ステージまたは3段パイプラインと呼ぶ．3段パイプラインは，単純に考えれば，パイプラインを行わない場合の3倍の速度で命令の処理をすることができる．しかし，実際の32ビットMPUのパイプラインはそれ以上の5段〜7段のパイプラインを行っており，非常に高性能を実現している．なお，パイプライン制御を行うMPUでは「命令フェッチ」のことを「先取り」をするという意味からプリフェッチと呼ぶ．

ところで，パイプライン制御は流れ作業のバケツリレーみたいなものであるから，どこかの段階で乱れが生じると，その処理効率が極端に低下する．MPUの世界では，パイプラインの乱れは主として分岐命令の実行時に引き起こされる．分岐命令が実行されると，そのときに「フェッチ」，「デコード」している命令がむだになり，分岐先からフェッチし直さなければならないので性能が低下するのである．

このため，分岐命令の高速化と分岐によるパイプラインの乱れの早期回復は，パイプライン制御を行うMPUの課題の一つとなっている．その解答として，最近では分岐先バッファや分岐予測機構を採用するMPUも登場しているが，これらの技術もまた大型計算機の流れを汲むものである．

(2) キャッシュメモリの内蔵

初めて大型計算機に仮想記憶が採用された当時と比較すれば，現在はメモリの値段が格段に安くなっている．しかし，32ビットMPUが最大性能を出すために想定されている高速メモリは，まだまだ高価だ．MPUに実力があっても，MPUがメモリの速度に足を引っ張られていたのではせっかくの32ビットの性能も活かせない．そこでMPU内に，たとえ少量でも高速なメモリを備え，外部の主記憶の内容をMPU内のメモリにコピーして持つというアイデアが考案された．これがキャッシュメモリである．

大型計算機の世界では常識だったキャッシュメモリも，MPUに実装するのはかなり困難だったようで，初期の32ビットMPU(68020やZ8000)では256バイト程度の容量しかもつことができなかった．しかし，MPUが進化するにつれ，1Kバイト，2Kバイトの容量は当たり前になり，中期の32ビットMPUのi486や68040では8Kバイトの容量をもつようになった．こうなると，MPUのアーキテクチャにパラダイムシフトが発生する．すなわち，遅いメモリのため命令フェッチのバンド幅が不充分だったので，1命令に多機能をもたせることを余儀なくされたCISCから，高速な命令供給に物を言わせて単純な機能しかない命令を短時間に多量に実行するRISCへの移行が始まったのである．

なお，キャッシュメモリといっても，さすがに256バイトでは「ないよりはまし」といった程度だが，8Kバイトとなるとかなりの手応えがある．仮想記憶のスワッピングの単位が4Kバイトであることが多い現状を考慮すれば，8Kバイトという容量はプログラム実行単位(サブルーチンやモジュール)を格納するのに十分な量ということができる．

現状のLSIの集積技術では，256Kバイト以上(昔の1000倍)のキャッシュメモリを内蔵することも可能になっており，隔世の感がある．

● RISCの台頭

1980年頃，米国スタンフォード大学とカリフォルニア大学のバークレー校において，RISCの研究が行われていた．RISCとは，Reduced Instruction Set Computer(縮小命令セットコンピュータ)の略であり，命令体系を単純化することで，それを実行するハードウェアも単純化し，高い動作周波数で高性能を得るという思想に則ったコンピュータのことである．スタンフォード大学やバークレー校の研究もその例から漏れない．その共通するアーキテクチャは，次のようなものだった．

- ロード/ストアアーキテクチャ
- パイプラインを用い，命令を1サイクルで実行
- 遅延分岐

これは，現在のRISCチップの特徴でもある．RISCでは，インタロック(パイプラインのステージ間の待ち合わせ)などの複雑な制御をハードウェアで行わないことが基本である．それによって，ソフトウェアが複雑になっても，ハードウェアをできるだけ簡単にすることを第一としていた．これは，コンパイラ技術の劇的な進歩をよんだ．RISCが使い物になると世間に認められたのは，コンパイラ技術の向上によるところが大きい．

コンパイラ技術に支えられたRISCの性能は，驚く

べきものだった．たとえば，MIPSの最初のRISCであるR2000は1986年に発表されたが，それを採用したSilicon Graphics社のグラフィクスワークステーションはわずか8MHzという低い動作周波数にもかかわらず，当時の32ビットMPUを採用したEWS(エンジニアリング・ワークステーション)以上の性能を発揮していた．

RISCの高性能は徐々に世間に認められるようになり，現在ではほとんどすべてのMPUがRISCになっている．IntelやAMDが開発しているx86プロセッサも命令体系自体は従来どおりのCISCのものを採用しているが，その中身はRISC技術を最大限に採用して高速化を実現したものだ．

● 新しいハードウェア技術

(1) スーパースカラとアウトオブオーダー実行

MPUがパイプラインを効率的に実現するようになると，1クロックで実行できる命令数が1に近づいてきた．MPUの命令は動作クロックに同期して処理されるため，1クロックで1命令の実行というのが限界である．パイプラインが1系統しかないので，1クロック1命令という壁ができるのである．

しかし，技術者の夢には際限がない．パイプラインを複数系統もち，それらを同時に実行するような構成にすれば，1クロックで1命令以上の処理を実現できる．これがスーパースカラである．

最近のMPUの性能競争のあおりを受けてか，高性能を謳うMPUはスーパースカラ構成が当然のようになっている．複数の命令を同時に実行するためには，命令間に入出力の依存性があってはいけない．その場合は，せっかく複数のパイプラインがあっても，1系統のみの実行となる．

しかし，そのような制限に甘んじていたのでは進歩はない．最初の2命令に依存性があっても，3命令目，4命令目との依存関係を調べると，依存性がなく，同時実行可能な場合がある．そのような条件を自動判別して，プログラムに書かれた命令の順序を無視して実行する方式が考えられた．もちろん，全体的に見て，逐次的に命令を実行する場合と比較して，結果に矛盾がないように考慮されている．これがアウトオブオーダー実行である．

(2) スーパーパイプラインと分岐予測

MPUの処理性能を上げる方法でもっとも簡単に思い付くのは，MPUを動作させるクロック周波数を高速化する方法である．しかし，動作クロックを向上させようと思っても，電気信号の伝わる速度というどうしても越えられない壁がある．

MPUの内部は無数の論理素子で構成されているが，その素子を電気信号が通過するたびに遅延が生じる．MPUの論理設計は，1クロックの間に所定の数の論理素子を通過するものと仮定して行われる．しかし，動作クロックが高速になると，1クロックの間に電気信号が通過できる素子数が限られてくる．もし，その素子数が設計値よりも多くなると正常に動作をしない．動作クロックとは，パイプラインを1段処理するための時間を規定するものである．このため，動作周波数が高くなるにつれ，パイプライン1段当たりの処理数が限られてくる．

この論理を逆手に取った手法がスーパーパイプラインである．つまり，処理単位を従来よりも細分化し，それらをパイプラインの1段に割り当てる．従来と同じ処理をするためにはパイプラインの段数を増やさなければならないが，それだけ速い動作クロックで回路を動かすことが可能になれば，パイプライン処理を行っている限り，1命令の処理時間は1クロックに見えるので，性能が低下することなく高速化を実現できる．

しかし，現実は甘くない．パイプライン処理で1命令の処理時間が1クロックになるのは，分岐がまったく存在しない場合である．もちろん，そのようなプログラムはほとんど存在しない．

分岐するかしないかは，命令の実行段階でなければわからないので，パイプライン動作を乱さないために分岐命令が実行段階になるまで，次のアドレスの命令フェッチやデコード動作は続けられる．つまり，分岐命令を実行する段階では，分岐命令の次の命令がデコード段階に，次の次の命令がフェッチ段階に入っているわけである．ここでの分岐命令の実行が「分岐する」だった場合は，新しく分岐先のアドレスで，パイプラインの最初の命令フェッチからやり直さなければならない．

パイプラインの段数が増加すると，分岐が発生した場合に分岐先の命令を新たにフェッチしデコードするための処理段数が増加することになり，それまでフェッチ，デコードした命令がむだになる．そのための時間的な損失は，最悪ならばパイプラインの総段数を実行するだけの時間にほぼ等しくなる．これでは，いくら周波数を上げて高速化しても実質的な性能に寄与しない．

そこで考えられたのが分岐予測である．それまでの方法は，常に分岐が成立しないものとして予測していると言えないこともない．それを分岐予測により，命令フェッチを行う方向を決定する．分岐予測が当たれば，パイプライン処理の損失はない．外れた場合はしかたないと割り切る．そのため，MPUのメーカーは分岐予測がいかに正確に行えるかという技術に心血を注いでいるのである．

(3) 投機実行

どんなにスーパースカラとパイプライン技術を駆使してMPUの性能を向上させたとしても，分岐予測の効率が性能を決定するキーポイントとなっていることに変わりない．しかし，分岐条件が未確定の間，命令のフェッチやデコードまでは可能だが，それを実行してレジスタなどを更新してしまったのでは，分岐予測が外れた場合に取り返しのつかないことになる．

単純な分岐予測だけでは，分岐条件が未確定な間は命令を実行ステージに移行することができないのである．これではパイプライン処理に乱れが生じてしまう．つまり，性能が低下することになる．

これを解決するための技術が投機実行である．要するに，分岐条件が確定しない場合でも予測した分岐方向にしたがって命令を実行し，結果を一時的な領域に退避しておく．そして，分岐予測が成功した場合に初めて一括してレジスタなどの更新を行う．投機的に実行した部分は，分岐予測が失敗すればむだになってしまうが，分岐予測が成功した場合は分岐条件確定までの待ち時間がなくなる．これにより，パイプライン処理を真に効率的に実行できるようになる．

最新の高性能MPUでは，スーパースカラ，スーパーパイプライン，投機実行のすべてを実現している．

● 64ビットへの道

MPUのアドレス空間が4Gバイト(32ビット)もあれば十分と考えられた時代も束の間であり，1990年代の初頭には4Gバイトでは不足するアプリケーションが登場するようになった．それを受けて，64ビットMPUであるDECのAlpha21064やMIPS社のR4000が登場した．

しかし，世の中の一般的なユーザーにおける32ビットから64ビットへの移行は，DECやMIPSの意向に反して，非常に緩やかなペースで進んでいるようである．2000年になって，ようやく64ビット化への兆しが見え始めた．

具体的には，ネットワークやデータベース関連の応用分野で扱うデータ量が巨大になり過ぎて，アドレス空間が32ビットでは不足してきたのだ．

とはいえ，32ビットから64ビットへの移行は，オーバーロードからオーバーマインドへの進化のように，進化そのものが目標であるような気がしないでもない．ましてや，128ビットというのは本当に必要なのだろうか．ともかく，64ビットプロセッサの真価が発揮されるまでにはもう暫くの時間が必要なのではないかと考える．やはり，当分は32ビットプロセッサが主流であることには変わりないであろう．32ビットプロセッサで，パイプライン，キャッシュ，仮想記憶といった，アーキテクチャが完成されたことを考えると，64ビット，128ビットの時代になっても，32ビットプロセッサは生き続けるのではないだろうか．

ただし，64ビット長や128ビット長のデータを扱うSIMD命令は，マルチメディア処理を行うために流行することが予想される．その場合，MPUのレジスタ長は必然的に64ビットや128ビットになるが，それをもって，64ビットや128ビットプロセッサと称するわけにはいかないと思う．

● 性能向上の最後の砦，マルチプロセッサ

人々は，MPUの性能向上の進歩以上の性能を要求する．それに応えるために，プログラムを並列実行可能な複数の部分(スレッドという)に分解して，それを複数のMPUで同時に実行しようとする考えがある．これがマルチプロセッサである．マルチプロセッサ自体はそれほど新しい考え方ではなく，大型計算機ではもちろん，サーバーなどの高性能を謳うコンピュータでは当然のように採用されてきた．

最近は，このマルチプロセッサを1チップ上で実現するというものがある．CMP(Chip Multi-Processor)という新語もできたほどである．面積の制限があるので1チップに集積されるMPUは2～4個である場合が多いが，将来的にはもっと多くのMPUが集積されることは想像に難くない．1チップマルチプロセッサの発想は十数年前から存在するが，実際のMPUに応用され始めたのは2001年になってからである．

わざわざ複数のMPUを一つに集積しなくても，一つのMPU内に同時実行可能な演算器を複数個持たせれば事が足りるという意見もある．これはスーパースカラの考え方である．

マルチプロセッサかスーパースカラか．こうなると

思想の問題である．考え方自体はOS論でのモノリシックカーネルとマイクロカーネルの議論に似てくるような気がする．単純にいうと，中央集権か地方分権かということであるが，それぞれの利点，欠点があり，どちらがよいとは一概にはいえない．

また，スーパースカラと1チップマルチプロセッサの中間的な解というものもある．それがマルチスレッディングである．Intelの猛烈なキャンペーンのせいで，最近ではハイパースレッディングと言ったほうが通りがいい．

これは一つのMPUで複数のスレッドを同時実行可能にするものである．つまり，プロセッサ内の演算器やキャッシュといったハードウェア資源を，時分割で複数のプロセスに割り当てて並列実行する．必要なハードウェア量と付随する性能の効率を考えると，この仕組みが主流になるかどうかは不明だが，マルチスレッディングを実現するMPUもいくつか発表されている．今後の推移に関しては，歴史の審判を待ちたい．

まとめ

RISCをとりまくプロセッサの歴史については，エピローグにまとめたのでそちらを参照してほしい．

人類は，その夢と理想をマイクロプロセッサという数ミリ角のチップに詰め込んできた．約30年前に初めて発表されたマイクロプロセッサは4ビットの処理能力しかなかったが，マイクロプロセッサは8ビット，16ビット，32ビット，64ビットと性能向上を達成してきた．いま，コンピュータの世界では32ビットが常識で，64ビットへの転換期にある．

ここ数年の動向を眺めていると，かつての大型計算機のMPUはEWSのMPUに駆逐され，そのEWSのMPUは(POWERやSPARCが気を吐いているものの)PCのMPUであったx86系MPUにとって代わられようとしている．これは，PCもEWSも性能差がなくなってきていることを意味する．MPUの発展は，始まったばかりなのか絶頂期なのか……神ならぬ身の知る由もなし．

アカルサハ　ホロビノ姿デアラウカ
人モ家モ　暗イウチハマダ　滅亡セヌ

太宰治『右大臣実朝』より

参考文献

1) T・R・リード，『チップに組み込め！マイクロエレクトロニクス革命をもたらした男たち』，草思社，1986年．
2) 嶋正利，『マイクロコンピュータの誕生　わが青春の4004』，岩波書店，1987年．
3) 星野力，『だれがどうやってコンピュータを創ったのか？』，共立出版，1995年．
4) 伊藤智義，久保田眞二，『BRAINS－コンピュータに賭けた男たち－(1)』，集英社，1996年．
5) 嶋正利，「技術開発と教育」，『Interface』，2002年6月号．

第1章

プロセッサの構成要素と動作の基本

プロセッサの基礎知識

プロセッサとは何だろう．専門書や教科書を読んでも難しそうだ．しかしMPUとは，そんなに複雑なものだろうか．じつは，その背景にある考え方は，単純なのではないだろうか．直感で理解できたら嬉しい．それが，本書のテーマである．まずは，MPUの基本的な動作について解説する．

1 コンピュータができること

● 数値計算

大規模数値計算，高度な意志決定支援，人工知能などコンピュータの応用分野は無限にある．しかし，コンピュータ自体ができることは単純である．データの「転送」，「加減乗除」，「論理演算」，「シフト」，「分岐」といった基本操作だけである．その計算結果を，人間が意味付けすること，たとえば，「結果がある値になったらある事象が成立したとみなす」とすることで，さまざまな結論を導き出す．つまり，ある一般的な問題を数値計算の操作に代表させ（これを問題のモデル化という），問題解決を行う．

極端な一例を示す．たとえば「命」と名付けられた記憶領域があり，そこには0または1の値が入るものとする．0は死んだこと，1は生きていることと意味付ける．コンピュータでの操作としたら，起動時に「命」に1を格納し，ある時間が経過したら「命」に0を格納するものとする．これは人の一生を計算していることになる．定期的に「命」の値を調べていれば，アバレンジャーのリジェのごとく，「あっ，死んだ」とか「あっ，生き返った」とか言うことができる．

あるいは，論理的な思考を実現するためには，仮定と結論の組み合わせやある項目から連想される別の項目を多数記憶しておき，最初の仮定から始まって，そこから得られる結論を次々と連想される項目に置き換えていくことで，最終的な結論を得られる．この操作を実現する基本操作は比較処理である．比較処理は，コンピュータでは排他的論理和という論理演算で実現される．また，連想結果の置き換えは転送処理に他ならない．結局，複雑に見える処理も基本操作の積み重ねで実現されるのである．その意味では，スーパーコンピュータのMPUも，EWSのMPUも，PCのMPUもできることに大差はない．違いは，データの処理速度くらいであろう．

● プログラム内蔵方式

さて，数値計算は定型的な処理であるので，ある程度自動化できる．たとえば，自動数表作成機や微積分装置などは，数値的な定型処理を機械化したものである．これらはコンピュータと呼べるだろうか．答えは否である．コンピュータをコンピュータたらしめる属性はプログラム内蔵方式と条件分岐にあるといわれている．

プログラム内蔵方式とは，コンピュータの動作を規定するプログラム（命令列）をシステムの記憶装置に内蔵していることをいう．命令とデータに区別はなく，命令の実行によって記憶装置にある命令を変更してそれを実行できるのが大きな特長である．このような命令の自己書き換えに関しては，デバッグの難しさや保護の難しさから推奨されない．しかし，これは粒度（書き換えてから実行するまでの時間的空間的距離）の小さい場合である．粒度の大きい場合は，ハードディスクなどの補助記憶装置から記憶装置に命令やデータをロードして，その部分を実行することに等しく，これはコンピュータシステムにおいてごく普通に行われる．

条件分岐とは，途中の計算結果に応じて処理を切り

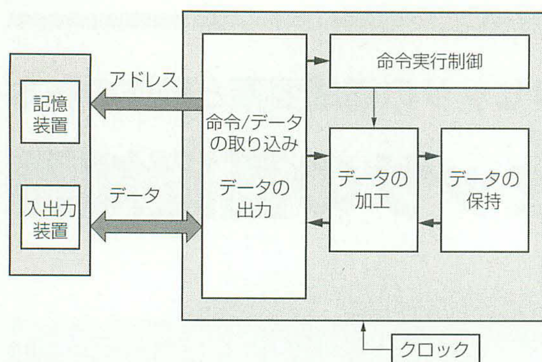


図1 MPUの構成要素

分けることができる機能を指す．プログラムを見ただけでは分岐が発生するか否かは不明である．その条件分岐を処理する段階になって初めて分岐するか否かが決定されるのである．条件分岐のおかげで，繰り返しや複雑な制御構造が実現できる．

以上のコンピュータの属性を一言でいえば，記憶装置にある命令を実行する有限オートマトンである．有限オートマトンの意味に関しては，抽象的で難しい概念なので，ここでは省略する．

そして，マイクロプロセッサ，あるいは，MPUとはコンピュータを1チップに集積したものである．本章ではMPUの具体的な動作原理について説明する．本章では主として「MPU」という単語を使用するが，ここでの議論は，それを「コンピュータ」に置き換えても，そのまま当てはまる．

2 MPUの構成要素

現代のMPUは**フォン・ノイマン型**と呼ばれる．これは**プログラム内蔵方式**のことであり，フォン・ノイマンが提唱したということになっているが，最近ではそれは誤りとされている．フォン・ノイマン型とかフォン・ノイマンボトルネックという言葉はやがて消滅するかもしれないが，ここでは慣例にしたがっておこう．

典型的なMPUは，「記憶装置」，「命令やデータを取り込むしくみ」，「命令実行を制御するしくみ」，「データを加工(処理)するしくみ」を基本的な構成要素とする．また周辺機器とデータをやりとりするための「入出力処理」というものもある．**図1**に典型的なMPUの構成要素を示す．

● プログラム内蔵方式には記憶装置が必須

プログラム内蔵方式には，プログラムを内蔵するための**記憶装置**が必須である．ほかの構成要素はMPUに内蔵されるが，記憶装置は，基本的にMPUの外部にある．この記憶装置は**メモリ**と呼ばれ，その構成によって**ROM**(Read Only Memory)と**RAM**(Random Access Memory)に大別される．

メモリとは複数の保存場所の集合で，各保存場所には位置を特定するための**アドレス**が付けられている．そして，あるアドレスを指定すると，それに対応する保存場所の内容が外部に読み出されるという装置である．RAMでは，アドレスと新しいデータを与えるこ

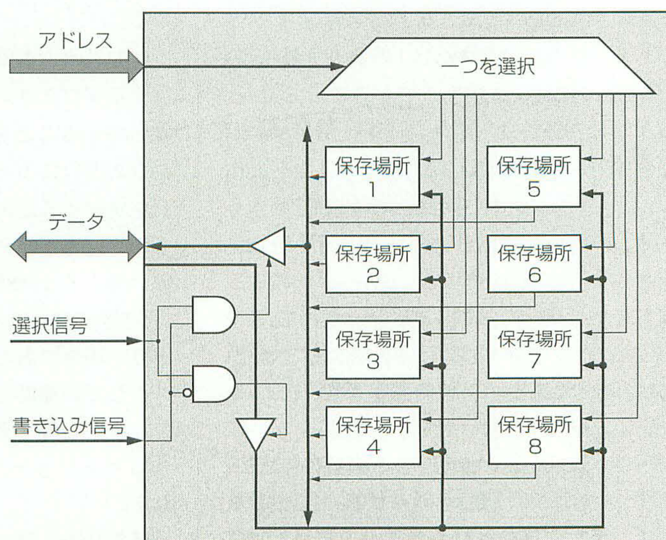


図2 メモリの概念図

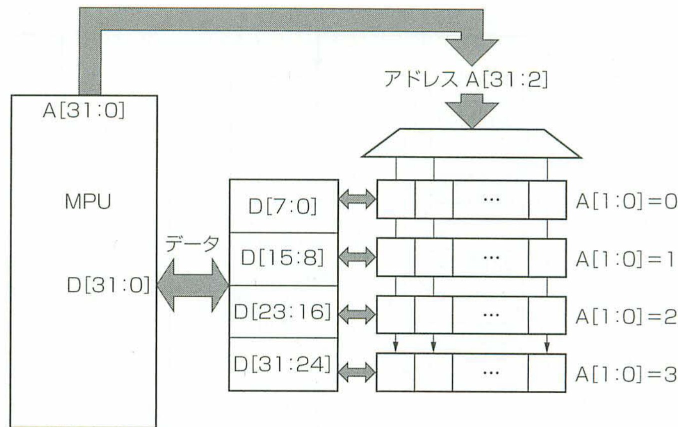


図3　32ビットデータバス

とで，アドレスで指定される保存場所の内容を変更することもできる．**図2**にメモリの概念図を示す．

メモリ内の保存場所の大きさ(ビット数)はいくつでもかまわない．しかし，最近のメモリは一つの保存場所の大きさを8ビット(＝1バイト)とする**バイトアドレス方式**が主流である．つまり，一つのアドレスを与えると1バイトのデータを得ることができる[注1]．

しかし，MPUの扱うデータは1バイトのみとは限らない．ハーフワード(16ビット)，ワード(32ビット)，ダブルワード(64ビット)といったデータ[注2]も扱う．おそらく，もっとも多用されるデータ長はワード(16ビットまたは32ビット)であろう．「○○ワード」という表現は，ワードが基準になっていることを示す証拠である．このため，MPUはある程度まとまったビット数(あるいはワード単位)のデータをメモリから取り出すほうが効率的である．このため，MPUとメモリ間のデータのインターフェース(データバス)は16ビット幅または32ビット幅であることが多い(アドレスのビット幅はまちまち)．たとえば，データバスが32ビット幅の場合は，1バイト出力のメモリを4個並列につなげて32ビットのデータ供給を実現できる(**図3**)．

アドレスはバイト位置を示すものである[注3]のに，一つのアドレスから4バイト(32ビット)のデータを取り出そうとすると「バイト並び(エンディアン)」の問題が生じる．複数バイトから構成されるデータに対し，バイトアドレスのより小さい保存場所にデータの上位の値を置くか，データの下位の値を置くかで2通りの方法がある．アドレスの小さい場所にデータの上位を置くのが**ビッグエンディアン**，逆にデータの下位を置くのが**リトルエンディアン**である．メモリでのバイト並びは異なるが，ビッグエンディアンでもリトルエンディアンでも，データバス上では同じイメージになる(**図4**)．

MPU内の演算器などはデータの下位側から計算をしていくので，その意味で，すべてはリトルエンディアンに集約されるといえなくもない．エンディアンとは，あくまでもメモリ上にデータがどの順序で格納されているかを示しているにすぎず，MPUの内部処理とは直接は関係ない．また，ビッグエンディアンの場合，ビット番号の名付け方がリトルエンディアンと逆順になっていることが多く，惑わされやすいが，実質(バイト内のビットイメージ)は同じである．

● 命令やデータを取り込むしくみ

MPUがまず行わなければならないことは，メモリに格納された命令やデータを内部に取り込むことである．そのためには，メモリに与えるアドレスを生成し，メモリから出力されたデータを取り込めばよい．

注1：メモリによっては保存場所の大きさが16ビット(×16という)，32ビット(×32という)のものも存在する．どちらかといえば，16ビットが一般的かもしれない．だたし，その場合もバイト単位での書き込みは可能になっている．

注2：インテルやモトローラに代表されるCISC時代からMPUを使ってきている人は，16ビットをワード，32ビットをダブルワード，64ビットをクオドワードと呼ぶ．現在は32ビットMPUが主流なので32ビットをワードと呼ぶのが自然だが，16ビットMPU時代の慣習も根強く残っている．CISCメーカーは16ビットをワード，RISCメーカーは32ビットをワードと呼ぶ傾向が強い．

注3：昔の大型計算機などではアドレスの割り付けがワード単位になっているものもある．つまり，すべてのデータはワード単位でしか扱わないのが基本である．このような場合にはエンディアンの問題はない．

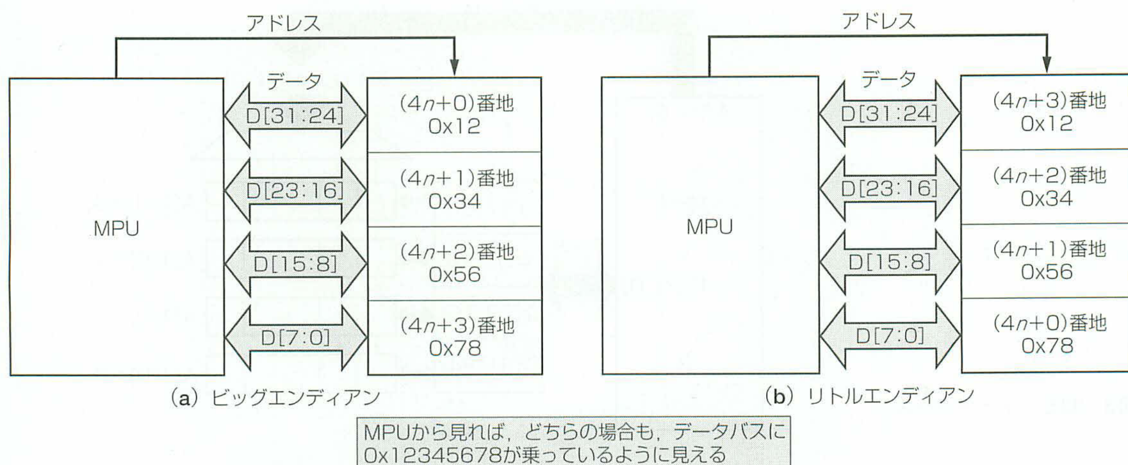


図4 エンディアンとデータバスの関係

まず，命令について考えよう．MPUは**PC**(Program Counter：プログラムカウンタ)という記憶機構(レジスタ)を備えている．これは，命令を取り込むメモリのアドレスを保持する．PCの値はアドレスバスを通じてMPUの外部に出力され，これがメモリに入力される．アドレスバスに値を出力すると，データバスを入力状態にしてメモリから出力された値(命令)を取り込む．これを**命令フェッチ**という．

MPUがリセットされると，ある特定の値がPCの初期値として設定される．そしてPCは，通常はメモリから取り込んだ命令のバイト数だけ値が増加していく．メモリから取り込んだ命令が分岐命令だった場合は，分岐先のアドレスが新たなPCとして設定される[注4]．

データバスから取り込まれた命令は，一般に，命令レジスタと呼ばれる記憶機構に保持される．それ以降の命令処理は，命令レジスタの内容にしたがって行われる．

さて，命令が扱うデータについて考えよう．メモリには命令のほかにデータも格納されている．命令によっては，その実行のためにメモリのデータが必要である．このため，命令の実行にともなって，メモリに格納された値が必要になった場合に活性化される機構を備えている．メモリを参照するアドレスはロード命令

注4：分岐先のアドレスは，通常，分岐命令の実行によって決定される．つまり，PCを分岐先に設定し直すタイミングは，分岐命令の実行後である．しかし，最近のMPUでは命令のフェッチと実行部分が切り離されている場合もあり，この場合は命令フェッチ部が自律的に分岐命令を処理する．

Column エンディアンの由来

エンディアンという言葉はコンピュータ用語の中ではそれほど古くない．1980年にDanny Cohenが『On Holy Wars and a Plea for Peace』という論文の中で初めて使用したというのが定説である．その語源はジョナサン・スイフトの『ガリバー旅行記』にある．小人国(リリパット)の中に出てくる，ゆで玉子を小さい端から食べる(割る？)主義の人々と大きい端から食べる主義の人々が由来である．「端」を表す「エンド」という単語に，「主義者」を表す「イアン」(例としてベジタリアンなどがある)が合成されてできた．

昔は，リトルエンディアンをインテル形式，ビッグエンディアンをモトローラ形式と呼んでいた．本来，リトルエンディアンはDECが，ビッグエンディアンはIBMが提唱したものらしいが．エンディアンのことを，ちょっと前のMIPSのドキュメントではSEX(性別)と書かれていた時期もある．なぜか，ビッグエンディアンが男(male)で，リトルエンディアンが女(female)である．

エンディアンという表現は，日本では坂村健氏が広めたような覚えがある．

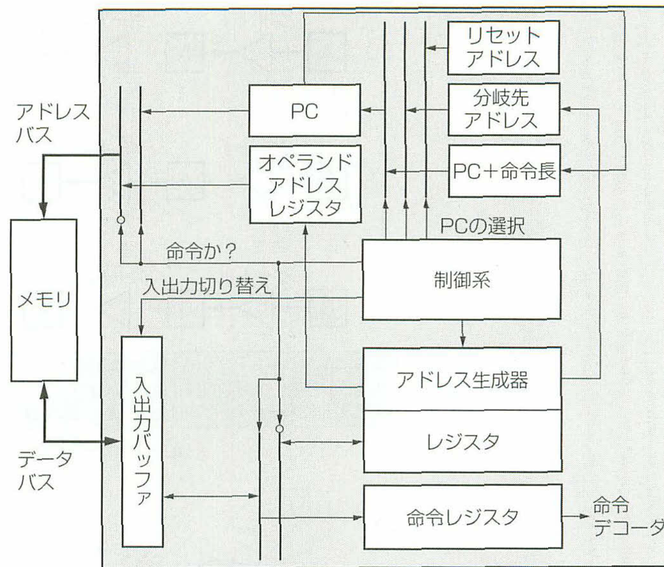


図5　命令やデータを取り込むしくみ

やストア命令などを実行(正確にはデコード)することで生成される．このアドレスは**オペランドアドレスレジスタ**(とりあえず，そう命名する)という記憶機構に保持される．オペランドアドレスレジスタの値はアドレスバスを通じてMPUの外部に出力され，これがメモリに入力される．アドレスバスに値を出力すると，データバスを入力状態にしてメモリから出力された値(データ)を取り込む．これを**オペランドフェッチ**，または，**オペランドリード**という．

図5にMPUの命令やデータを取り込むしくみを示す．

● 命令実行を制御するしくみ

MPUは，メモリから取り込んだ命令を解釈し実行する．メモリから取り込まれた命令は**命令レジスタ**に保持され，それが**命令デコーダ**によって解釈(デコード)される．デコードとは命令コードに含まれているMPUに対する指令を取り出す機構である．具体的には，命令の種類を判別し，取り出した情報に基づいて，実行すべき演算の種類を決定したり，必要な制御信号を生成したりする．

命令がデコードされると，命令ごとにその後の処理手順が決定される．命令の実行とは，MPUの内部状態が変化することであり，ある状態になると回路が特定の状態(たとえば，入力を演算器に入れるとか)に変化する，またある状態では回路が別の状態(たとえば演算器の出力を取り出すとか)に変化する，ということを繰り返すことで実現される．この命令実行は，一般に，**クロック**と呼ばれる周期的に変化する信号に同期して行われる．クロックが進むにつれて，内部状態は，ある命令では，

S0→S1→S2→S3

と状態変化をし，また，ある命令では，

S0→S4→S5→S5

と状態変化をする．ここでいう内部状態とは，具体的には，MPU内の各ゲートをON/OFFする組み合わせを示す．ある内部状態は，命令デコード結果と命令実行の中間結果を受けて，次にどの内部状態になるかが決定される．この状態変化によって，MPU内をデータが流れていく〔**図6(a)**〕．

以上が命令デコード後の制御であるが，MPU全体も，

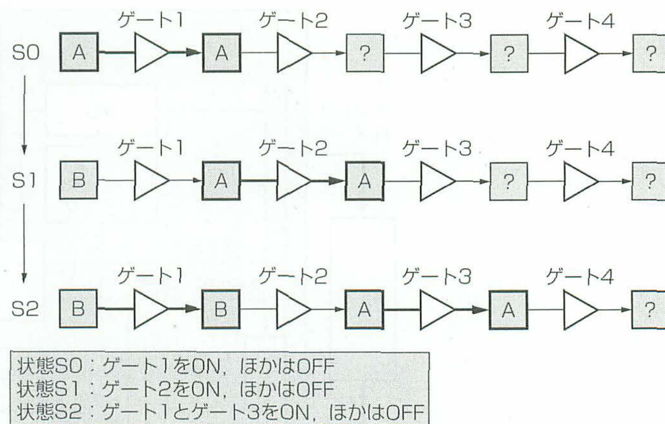
命令取り込み→命令デコード→命令実行

という大きな状態遷移をしながら制御されている〔**図6(b)**〕．

要するに，命令実行は状態遷移の塊なのである．このような状態遷移を司る機構を，**ステートマシン**，あるいは，**シーケンサ**と呼ぶ．つまり，MPUの実行とは，大小さまざまなシーケンサが組み合わされて複雑な状態遷移を行うことで実現されるのである．

● データを加工(処理)するしくみ

命令の実行とは，メモリに格納されているデータに対して何らかの加工(何もしないことを含む)をして，メモリに書き戻したり，命令実行の状態を変化させたりすることである．



(a) 状態変化とデータの流れ

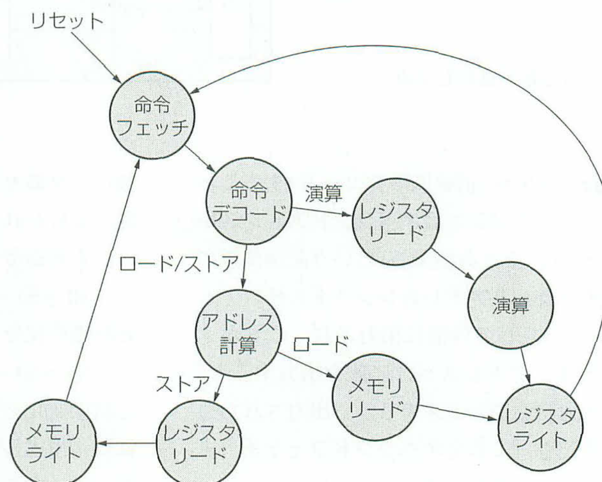


図6 MPU内部の状態遷移

(b) MPU全体の状態遷移例

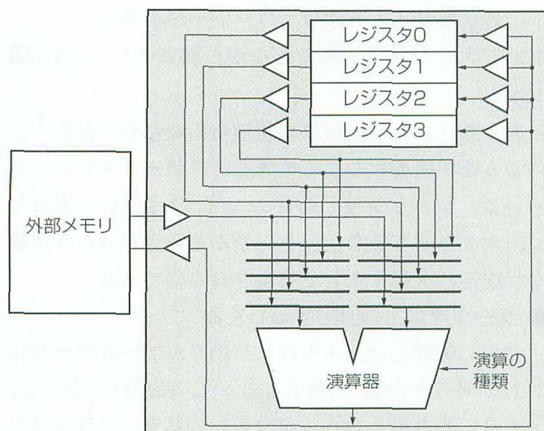


図7 データを加工するしくみ

メモリから取り込まれたデータは，レジスタと呼ばれるMPU内部のメモリに一時的に退避されることもある．MPUは，レジスタ(内部メモリ)やメモリ(外部メモリ)からのデータを**演算器**に適宜与えることでデータを加工する(**図7**)．

演算器のことを**ALU**(算術論理ユニット：Arithmetic Logic Unit)と呼ぶ．これは，加減算を行う算術ユニット(Arithmetic Unit)と論理和，論理積，排他的論理和などを行う論理ユニット(Logic Unit)の総称である．基本的な演算のうち，乗算と除算は専用のユニットで実現される．

どのメモリから演算器の入力をもってくるかという点は，MPUのアーキテクチャ(設計思想)に大きく関係する．基本的には，二つ(同一でもいい)のレジスタから入力データをもってくる．しかし，

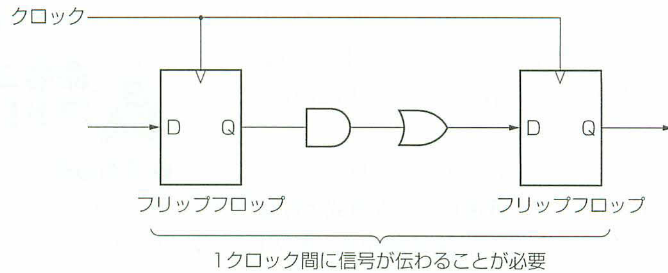


図8　回路の模式図

メモリ　　と　レジスタ
レジスタ　と　メモリ
メモリ　　と　メモリ

といった入力の組み合わせも当然考えることができる．ソフトウェアの作りやすさを考慮すれば，すべての組み合わせが可能なほうがプログラミングの自由度が高く嬉しい．ハードウェアの設計しやすさを考慮すれば，メモリから取り込んだデータは必ずレジスタに格納することとし(これは何もしないという処理)，演算器の入力はレジスタのみに限定したほうがハードウェア設計が楽で嬉しい．

前者はCISCの考え方であり，後者はRISCの考え方である．ただ，二つの入力がどちらもメモリというのは，制御がかなり複雑になるため，CISCであっても，入力の少なくとも一方はレジスタに限定されていることが多い．

演算器での演算の種類は，命令デコードによって決定される．なお，**図5**にあるアドレス生成器も演算器の一種であり，その実体は加算器である．アドレス生成器は，専用にもつ場合と，通常の演算器で代用する場合がある．

● クロック

MPUの動作を理解するとき，クロックの存在を忘れることはできない．クロックとは一定の周期で0と1を繰り返しながら自走している特殊な信号である．これは，MPUの動作タイミングの基準となる．1秒間に1回だけクロックが0から1に変化する(1周期)速さを1Hz(ヘルツ)という．最近のMPUは800MHzとか1GHzというクロックで動作するが，800MHzとは1秒間に800×M(メガ：100万)＝8億回，1GHzとは1秒間に1×G(ギガ：10億)＝10億回，クロックが変化することを意味する．

MPUの内部回路はフリップフロップの集合で構成される．フリップフロップとは，クロックの切り替わり時に新たな状態(0または1という信号)を保持する回路である．フリップフロップを複数個並列に並べたものがレジスタである．

MPUの回路を，フリップフロップとフリップフロップ間を配線したものとして模式化すると(**図8**)，回路が動作するということは，クロックが1回変化する間に前段のフリップフロップに保持した値が，導線を通じて，後段のフリップフロップに保持し直されることに相当する．レジスタなどを考える場合，フリップフロップ間の配線長は完全には同じでないので，後段のフリップフロップに信号が到達する時間は同一ではない．あるいは，フリップフロップ間には論理積や論理和のゲートが挿入されていて，ゲートを通過するごとに少しずつ信号が遅延する．クロックとは，その信号の到達時間を規定するものである．つまり，1クロック間に信号が伝わらないと誤動作する．全部の信号が正しく伝わるためのクロックの最高周期が，MPUの最高動作周波数である．

● 汎用レジスタ

MPU内部のメモリをレジスタということはすでに説明した[注5]．レジスタは演算器の入出力となるデータの一時記憶場所である．レジスタは演算器のデータとなるほかに，アドレッシングモードにおいて，レジスタ間接モード用のベースアドレスやインデックスを

注5：厳密にはレジスタとメモリは異なる．メモリにはフリップフロップで値を保持するSRAMとコンデンサの容量で値を保持するDRAMがあり，通常，MPU内部に搭載されるのはSRAM(キャッシュメモリなどに使用される)である．その意味で，レジスタもメモリもフリップフロップの集まりであるが，メモリは専用に設計され，小面積で大容量の情報を記憶できる．つまり，レジスタをSRAMで構成すれば面積が縮小できるのだが，SRAMはクロックに同期して動くわけではないので，アクセス時間を規定しにくく回路設計が難しくなる．

与えるためにも用いられる．このようにレジスタの用途はいろいろあるが，すべての用途に使用できるレジスタを**汎用レジスタ**(General Purpose Register)という．

最近のMPUの提供するレジスタは汎用である．このため，最近のMPUは，**汎用レジスタ方式**と呼ばれる．しかし，昔はハードウェアを簡略化するために，目的別の専用レジスタを用意するものもあった．つまり，ベースアドレス用，インデックス用というようにレジスタの用途が限られていた．このような場合，演算器の入出力となるレジスタも限られている．具体的には，演算器の入力の一つと出力が演算用レジスタ(アキュムレータ)に接続されている．演算器のもう片方の入力が汎用で，ほかの専用レジスタやメモリ，アキュムレータ自身に接続されている．このような方式は**アキュムレータ方式**と呼ばれる．

● 入出力

入出力とは，たとえばx86系のMPUでは`MOV`という転送命令のほかに`IN`や`OUT`という入出力のための命令が存在する．`MOV`はメモリやレジスタに対するデータの入出力を司るので，新たに入出力といわれてもピンとこない．ここでいう「入力」と「出力」とは周辺装置からの情報の入力，周辺装置への情報の出力を示す．しかし，これらの命令が実際に行う処理は，あるアドレスからデータを入力すること，あるいは，あるアドレスへデータを出力することである．それではますます，`MOV`との違いがわからない．実は「入出力」命令でのアドレスはメモリではなく，周辺装置に直接接続されているのである．メモリと区別する意味で，「入出力」命令でアドレスする(指し示す)対象を**I/Oポート**という．

通常の`MOV`命令と`IN/OUT`命令を区別する場合，アドレスとデータの入出力時には，それがメモリかI/Oポートであるかを示す信号(外部端子)が使われる．MPUを使ったシステムを構築する場合は，このメモリなのかI/Oなのかの信号を見て参照する対象を振り分けられるようにしなければならない．

MPUによっては専用の「入出力」命令を提供していないものもある．これはメモリ空間の特定位置をI/Oポートとみなす方式である．これを**メモリマップトI/O**という．

I/Oポートとメモリの差異は，その逐次性にある．メモリに対しては投機実行やアウトオブオーダー実行が考えられるが，I/Oにはそれがない．これらについては，後述する各章で解説する．

3 命令コード，オペランド，アドレッシングモード

● 命令の形式

ここではMPUが処理する命令に関して考える．上述のように，命令はデコードされることによって処理に関する情報を抽出する．逆にいえば，命令には処理に関する情報が符号化(エンコード)されている．命令は数値の形態でメモリに格納されている．この意味で命令とデータに区別はない．そして，命令を示す数値をビットで表すと，それぞれのビットが意味をもっている(情報がエンコードされている)ことがわかる．命令を示す数値を**命令コード**と呼ぶ．

一つの命令コードのビット数は何ビットでもかまわないのだが，メモリに効率良く格納できるように1バイト(8ビット)の倍数が用いられる．CISCでは命令の種類によって命令コードのビット長がバイト単位で可変になっていたりするが，RISCでは命令デコードを簡単に行うために固定長(16ビット，あるいは32ビット)であることが多い．

一般に命令コードは，**オペレーションコード**(オペコード)と**オペランド**の二つの領域に分けられる．オペコードは命令の種類を示し，オペランドは扱うデータの形態を示す．オペランドは**アドレッシングモード**で規定される．アドレッシングモードとは，データがどこに格納されているかを示す形式である．アドレッシングモードの例としては，

- 即値(イミディエート)
  命令コードに埋め込まれた定数値
- レジスタ
  レジスタにデータがある
- 直接アドレス
  アドレスで示すメモリにデータがある
- レジスタ間接
  レジスタにあるアドレス値であるメモリにデータがある
- インデックスつきレジスタ間接
  レジスタにあるアドレス値にインデックスレジスタの値を加えたアドレスにデータがある
- ディスプレースメント付きレジスタ間接
  レジスタにあるアドレス値にディスプレースメントを加えたアドレスにデータがある

などがある．メモリ参照は，基本的には，

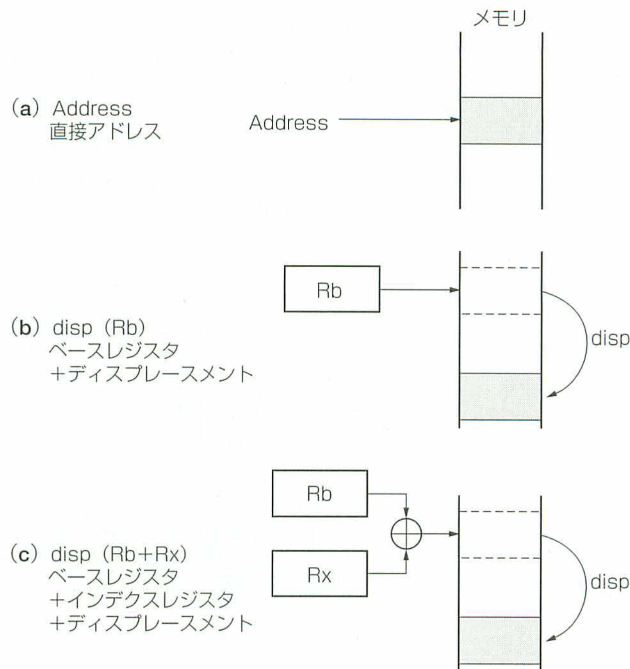


図9　メモリアドレッシングの例

ベースレジスタ＋インデックスレジスタ
＋ディスプレースメント(オフセット)

ですべてが表せる．このとき，インデックスレジスタの値は参照するデータサイズにしたがってスケーリングされる(バイトなら×1，16ビットなら×2，32ビットなら×4，64ビットなら×8)こともある．つまり，インデクスの値は何番目のデータであるかを示す．これを**自動スケーリング**という．

また，**メモリ間接アドレッシング**というものもあり，これは，上述のアドレス計算で求められたメモリの内容を新たなベースアドレスとし，ディスプレースメントを加えてメモリアドレスとするものである．

なお，ベースレジスタとしてプログラムカウンタ(PC)を指定できる場合もある．これを**PC相対アドレッシング**と呼び，ポジションインデペンデントなオブジェクトコードを作成する場合に使用される．**図9**にアドレッシングモードの例を示す．

このように，アドレッシングに関してはいろいろな方法が考えられるが，CISCは豊富なアドレッシングモードを特徴とし，RISCは単純なアドレッシングモードを特徴とする．具体的には，CISCは何でもありで，RISCは即値，レジスタ，ディスプレースメント付きレジスタ間接が典型的なアドレッシングモードである．

ところで，命令の実行は，二つのソースオペランドを入力とする演算を行い，結果をデスティネーションオペランドに格納する．二つのソースオペランドと一つのデスティネーションオペランドを独立に指定することができるのが**3オペランド方式**であり，一つのソースオペランドとデスティネーションオペランドが共通なものが**2オペランド方式**である．3オペランド方式では命令の中に三つのオペランド領域が存在し，2オペランド方式では命令の中に二つのオペランド領域が存在する．

オペコードのビット数は命令の個数を示す．つまり，2ビットなら4種類，3ビットならは8種類，4ビットならば16種類，5ビットならば32種類……，という具合である．命令コードのオペコード以外のビットはオペランドを示す．このビットが，基本的には，2オペランド方式では二つに，3オペランド方式では三つに分割される．オペランドのうち，アドレッシングモードに含まれるレジスタはレジスタの番号で示される．このレジスタ番号を示す領域のビット数は，MPUが備えるレジスタの本数によって決定される．4本なら2ビット，8本なら3ビット，16本なら4ビット，32本なら5ビット……，という具合にビットが必要である．また，オペランド領域の分割のやり方は，命令の種類やアドレッシングモードによって少しずつ異なる．この違いを**命令形式**(フォーマット)という．命令形式の例を**図10**に示す．

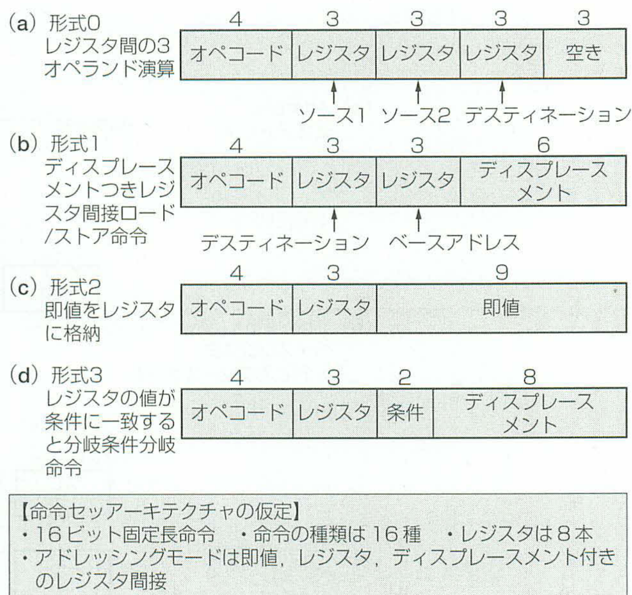


図10　命令形式の例

以上の説明からわかるように，命令コードのビット長は，オペコードの種類，アドレッシングモードの種類，レジスタの本数，オペランドの数などで決定される．これらを総称して**命令セットアーキテクチャ**という．MPUでは，無限に長い命令コードを使用できるわけではないので，その命令セットアーキテクチャで命令コード長が決定される．逆にいえば，固定長の命令コードを採用する場合は，命令セットアーキテクチャを詳細に検討しなければ，情報が命令コードに入りきらなくなってしまう．

● 命令の流れと実行

MPUの構成要素がわかったところで，それらがどのような係わりをもって動作するのかを見ていこう．**図11**に模式化したMPUのブロック図と，演算命令/ロード命令/ストア命令に関し，各状態での命令とデータの流れを太線で示す．もちろん，MPUにはこれら以外の命令も存在するが，ここでは省略する．命令フェッチと命令デコードはすべての命令で共通であるが，それ以降は命令デコード結果によって異なる状態遷移をする．

(1) 演算命令(レジスタ-レジスタ演算)

演算命令は，命令フェッチ〔**図11**(a)〕，命令デコード&レジスタリード〔**図11**(b)〕，命令実行〔**図11**(d)〕という状態遷移を行うことで命令を実行する．

(2) ロード命令

ロード命令は，命令フェッチ〔**図11**(a)〕，命令デコード〔**図11**(c)〕，アドレス計算〔**図11**(e)〕，メモリアクセス1＝オペランドリード〔**図11**(f)〕，メモリアクセス2＝レジスタライト〔**図11**(h)〕という状態遷移を行うことで命令を実行する．

(3) ストア命令

ロード命令は，命令フェッチ〔**図11**(a)〕，命令デコード〔**図11**(c)〕，アドレス計算〔**図11**(e)〕，メモリアクセス1＝レジスタリード〔**図11**(g)〕，メモリアクセス2＝オペランドライト〔**図11**(i)〕という状態遷移を行うことで命令を実行する．

● パイプライン

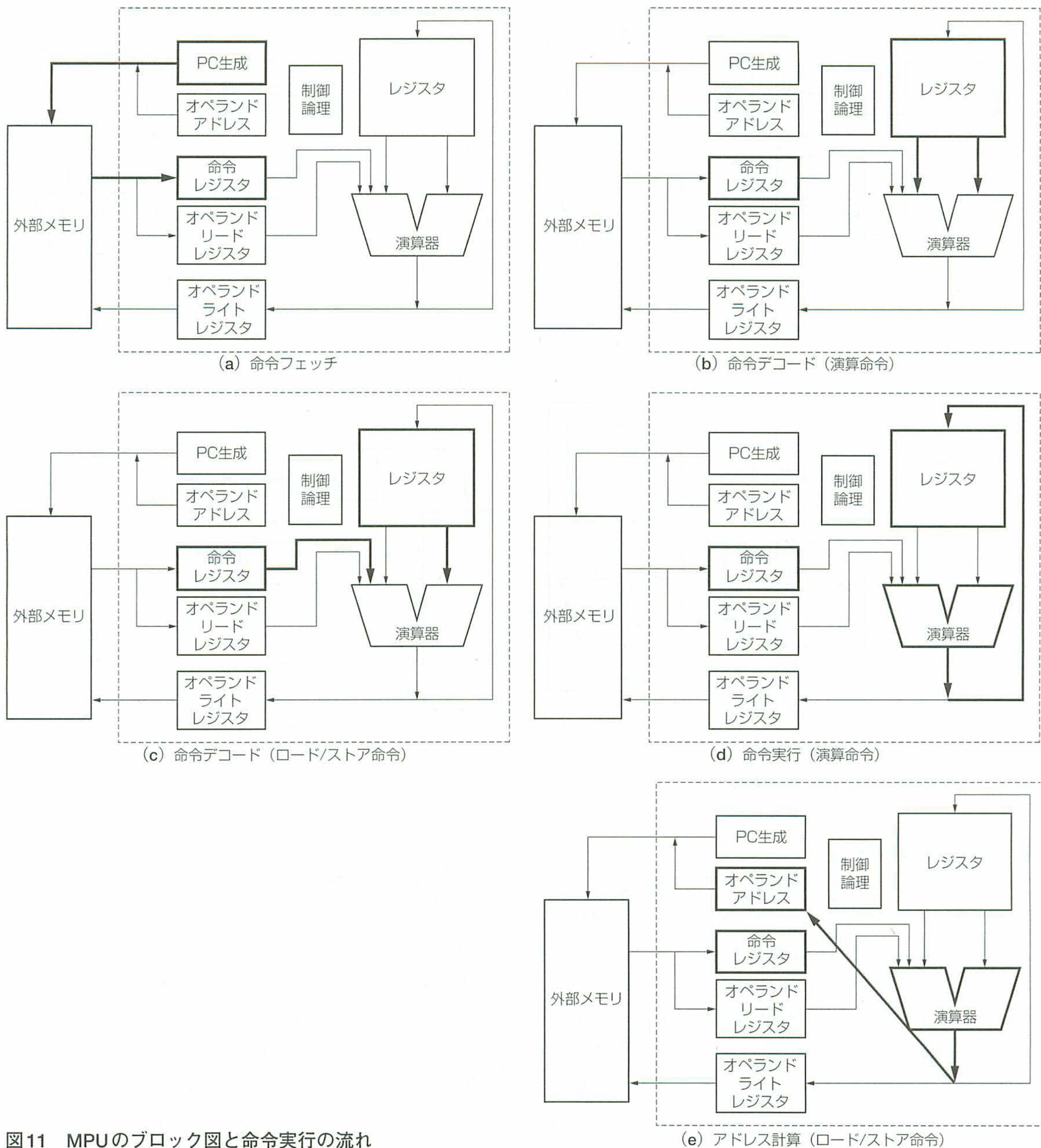
昔のMPUは一つの命令の実行が終わった後で，次の命令の実行を開始していた．しかし，命令実行に係わる状態遷移において，演算器など多くのハードウェア資源は1回しか使用されない．これでは，あまりにも効率が悪い．命令実行の状態をオーバラップさせることで，クロックごとにハードウェア資源を使用することができ，命令実行のスループット(クロックごとに実行が終了する命令の個数)も向上する．これがパイプラインの考え方である．

パイプライン構造でMPUを動作させるためには，各状態で演算結果などのデータを保持しておけばよい．**図12**(p.27)に，演算命令をパイプライン構造で実行する場合のハードウェア資源の使用状況を示す．

パイプラインに関しては，第2章で詳細に解説する．

● キャッシュ

これまでの説明は，メモリを1クロックで参照できるものとして話をすすめてきた．しかし，現実的には，



(a) 命令フェッチ
(b) 命令デコード（演算命令）
(c) 命令デコード（ロード/ストア命令）
(d) 命令実行（演算命令）
(e) アドレス計算（ロード/ストア命令）

図11 MPUのブロック図と命令実行の流れ

メモリのアクセス時間はMPUのクロックと比べて非常に遅い．実際の命令実行では，メモリを参照する状態（命令フェッチ，オペランドリード，オペランドライト）は複数のクロック数を消費する．これでは，命令実行がメモリのアクセス時間に律速されてしまい，高速な実行ができない．

そこで考案されたのが，従来は外部にあったメモリをMPUの内部に取り込むことである．この場合，高速なSRAMを集積することが可能になり，1～2クロックでメモリを参照できる．これが**キャッシュメモリ**である．しかし，外部メモリと同じ容量のキャッシュメモリを内蔵することは不可能である．そこで，外部メモリの一部分をコピーしてキャッシュメモリに格納し，外部メモリとの入れ替えを適宜行って，高速な命令実行を維持する．

キャッシュメモリに関しては，第4章で詳細に解説する．

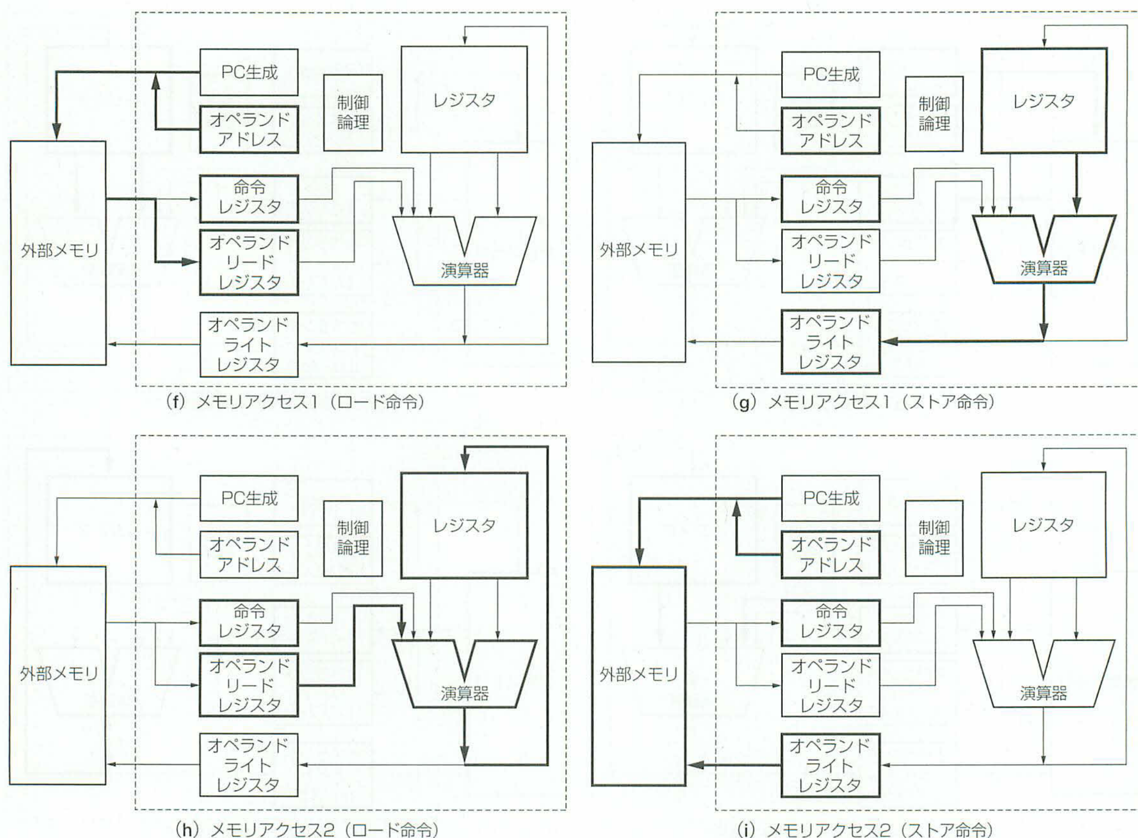


図11　MPUのブロック図と命令実行の流れ（つづき）

● MMUと仮想記憶

命令やデータを格納する外部メモリの容量には限界がある．しかし，ソフトウェアの高度化にともない，プログラムがメモリに格納しきれない場合も出てきた．これを解決するために，プログラムを分割してハードディスクなどの補助記憶装置に格納し，少しずつメモリの内容と置き換えながら，それを実行するという方式が考えられた．発想はキャッシュメモリと同じである．これを行うためには，プログラムで参照するアドレス（仮想アドレス）を実際のメモリのアドレス（物理アドレス）に変換する必要がある．この機能を提供するのが**MMU**（Memory Management Unit：メモリ管理ユニット）である．

MMUには，マルチタスクの実現，メモリ保護の実現という機能も有している．MMUに関しては，第5章で詳細に解説する．

● 割り込み

MPUの処理内容が高度化してくるとソフトウェアも複雑になる．そこで，プログラムの本筋とは直接関係のない処理を独立に行うという発想が生まれた．それを実現するのが割り込みである．割り込みが発生すると，MPUはそれまで実行していた処理をいったん中断し，割り込み処理という別の処理の実行を始める．割り込み処理が終了すると，プログラムの実行は割り込みが発生した時点から再開する．プログラムは本筋のプログラムと割り込み処理用のプログラムを独立に開発できる．割り込みを使用しない場合は，割り込み処理で行うような別処理を本筋のプログラムにサブルーチンコールをさせて実行しなければならず，プログラム開発において，そのための余計な考慮が強いられる．

割り込みに関しては，第6章で詳細に解説する．

● プログラムとは

MPUすなわちコンピュータは，メモリに格納された命令を取り込み，その指示するとおりに動作する．昔から，コンピュータはソフトウェアがなければ只の箱（粗大ゴミとも）といわれる．ソフトウェアとはプログラムのことであるが，それではプログラムとは何だろう．これは，本章の目的ではないので簡単に説明する．

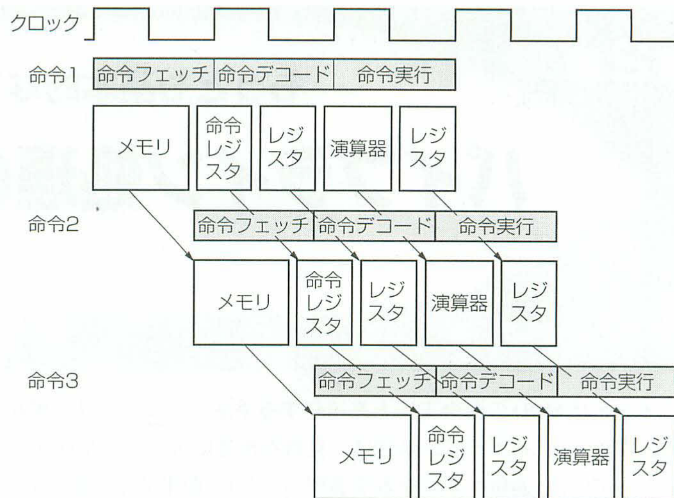


図12　パイプラインとハードウェア資源の供給

コンピュータが理解できる命令は，コンピュータごとに定義された**機械語**と呼ばれるビット列である．一般的には，ビット列が8ビット(バイト)，16ビット(ハーフワード)，32ビット(ワード)の塊になってメモリに格納されており(そのほうがメモリにとって都合がよいので)，それが機械語の命令として認識される．命令が参照するデータも命令と同じ形式をしており(区別がない)，それがコンピュータ自体の属性でもある．

命令をある規則にしたがって並べ，コンピュータに与えると，特定の仕事をさせることができる．プログラムとは，目的の仕事をコンピュータに実行させるために，それに最適な規則(アルゴリズムという)にしたがって，(機械語の)命令を並べたものを意味する．プログラムの実行時には,この命令列をメモリに格納し，コンピュータの動作を開始させる．すると，これまでに示した手順で命令が処理されていく．命令処理にはデータが必要なので，昔から，プログラムとはアルゴリズムとデータ構造を合わせたものである，とよく説明される．

さて，機械語は0と1のビット列なので人間には理解しにくい．それを人間が理解しやすくするために意味をもった(英語に近い)記号に対応させて扱う．つまり，MOV(転送)とかADD(加算)といった記号で機械語を代表させる.これらの記号を**ニーモニック**と呼ぶ.ニーモニックを使用してプログラムを書くための手段が**アセンブリ言語**である．アセンブリ言語を機械語に変換するしくみ(これも結局はプログラムである)が**アセンブラ**である．

アセンブリ言語は，人間に理解しやすくなっているとはいえ，所詮，機械語とほぼ1対1に対応しているものなので，機械語そのものである．個々の命令機能が単純すぎて複雑なアルゴリズムを記述するためには向いていない．そこで考案されたのが，人間の思考を反映させやすくした**高級言語**である．「高級」というのは「飛び抜けた」という意味ではない．言語仕様が機械語にどの程度近いかの指標であり，アセンブリ言語は低級言語といわれる．その対極としての「高級」である．

まとめ

この章を執筆していて，何だかコンピュータの教科書を作っているような幻想にみまわれた．コンピュータの教科書にはいろいろな知識が詰まっているが，その項目があまりにも多いので，ややもすると本質を見失いがちである．本書の目的はMPUのしくみを直感的に理解しようというものなので，2進数やブール代数といった一般の教科書に載っているような事項は解説しない．

本章は，いわば導入あるいは概説である．次章から各論に入る．

第2章

もっとも基本的なプロセッサ高速化技法

パイプライン処理の概念と実際

パイプラインとはMPUの命令実行を高速化する手法の一つであり，現在では，ほとんどすべてのMPUで採用されている．RISCのパイプライン処理は，見事なまでにヘネシー＆パターソンが提唱した5ステージのパイプラインにしたがっている(通称ヘネパタ本を参照のこと)．前半では一度に1命令を実行する通常のパイプライン(シングルパイプライン)についての基本概念を説明する．2命令以上を同時実行するものはスーパースカラと呼ぶが，これと対比する場合はユニスカラパイプライン，あるいは単にスカラパイプラインとも呼ばれることもあるようだ．後半ではシングルパイプラインの代表とも言えるR3000，SHシリーズ，ARM，V800シリーズのパイプライン構造を解説する．

パイプライン処理の概念

1 パイプラインとは

● 流れ作業＝パイプライン

コンピュータの性能を向上させる方法については，いろいろ考案されている．**パイプライン**とは，ハードウェアを並列化して性能を向上させるための一般的な手法である．その基本的な考え方は，プログラム内蔵方式を提唱したフォン・ノイマンによってすでに提案されていたという．たとえば，MPUの命令実行に比べて10倍以上も遅いメモリアクセスが存在する状況下で効率的に命令の処理を行うために，命令の実行とメモリアクセスをオーバラップして処理することが考えられた．これが，パイプライン処理の原型である．

パイプラインの基本的な考え方はごく自然なものである．なにもコンピュータの技術に固有なものではない．自動車の製造ラインや電子部品工場などで行われている流れ作業は，パイプラインそのものである．一つの製品が数分後ごとに完成していくようすを思い浮かべてほしい．実際，パイプラインの呼び名は，石油が次々とパイプを通過していく石油化学パイプラインと動作が似ていることに由来している．

各工程が1単位時間かかるN工程からなる処理を考える．単純に考えると，この処理を終了するためにはN時間を要する〔**図1(a)**〕．これをN人の人が流れ作業によって各工程を分担し，前の工程から受け継いだ製品に1単位時間で加工を施して，後の工程に引き継ぐようにする〔**図1(b)**〕．この場合，もともとの処理ではN時間に一つしか製品が完成しないが，流れ作業では見かけ上，1単位時間に一つの製品が完成することになる．つまり，処理速度はN倍に改善される．これがパイプラインの原理である．

● ステージ，段数，ハザード

ここで，各工程をパイプラインの**ステージ**という．「段」という表現も使われ，N工程から構成されるパイプラインはNステージパイプライン，またはN段パイプラインと呼ばれる．また，あるステージを分担する人が手間取って，そこでの処理を1単位時間以内に終わらせることができないような場合は，パイプライン処理に乱れが生じ，処理性能が低下する．パイプラインステージでの処理を単位時間内に終わらせることを阻害する要因を**ハザード**という．

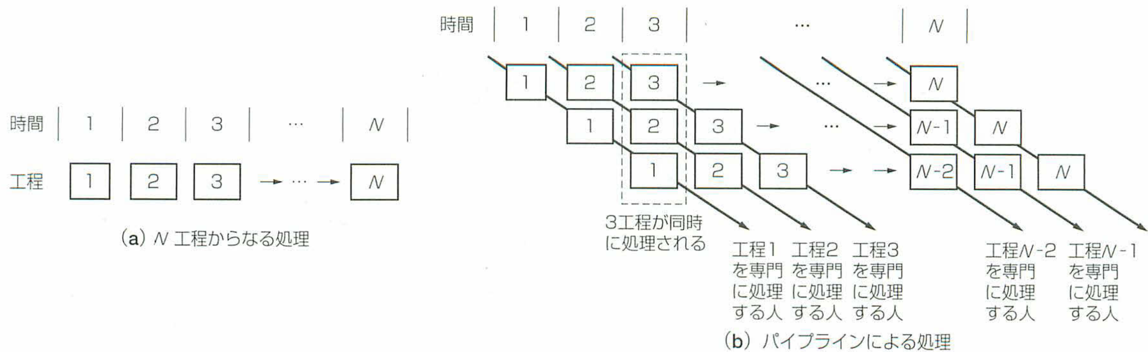


図1 パイプライン処理の概念

パイプライン処理をコンピュータに適用する場合は，各ステージが並列に処理できることが前提である．ハードウェア資源を共有するステージがあると，ハザードが生じ，待ち合わせが必要になる(これを**ストール**という)．逆にいうと，ハードウェア資源が競合しないようにパイプラインステージを分割するのがプロセッサ設計者の腕の見せどころである．

パイプライン処理は，まず大型計算機で採用された．その後，半導体の集積技術が進み，MPUでも大量のトランジスタが利用可能になると，MPUにも採用されるようになる．パイプライン処理の採用を大々的に表明したMPUは，NECのV60が最初ではないかと筆者は記憶している．それ以前のIntelの8086でもオペランドフェッチと実行をパイプライン化していたが，Intelがパイプラインを明言したのは80386以後(最近のIntelの発言ではPentium以降)となっている．一方，68000系のMPUも古くからバスサイクル同期のパイプライン処理をしていたようである．しかし，こちらもパイプラインを明言したのは68060が初めてだったと思う．68060はすでにスーパースカラ構造になっていたので，シングルパイプライン時代の68000系のパイプライン構造は不明である．

2 パイプラインの理論

● パイプラインステージ

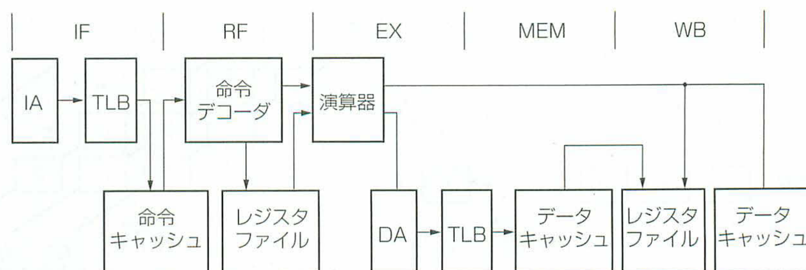
CISC初期においてもパイプライン構造を採用しているものがあった．しかし，それらのMPUにおいてパイプライン処理は有効に機能していたとはいえない．各MPUメーカーがパイプラインを強調しなかったのは，それが性能に寄与していなかったからではないかと考えられる．しかし，RISCの登場によってパイプライン処理はにわかに脚光を浴びる．RISCのパイプラインは，CISCとは異なり，全命令でパイプラインのステージ数は固定であることが多い．筆者だけの感覚かもしれないが，命令フェッチ，命令デコード，実行という処理の流れも，その区切りが明確になっているように感じる．

RISCの存在意義は，パイプライン処理をいかに効率的に実現できるかにかかっているといっても過言ではない．このため，RISCでは命令やオペランドをキャッシュからフェッチすることを前提としている．通常のメモリはアクセス時間が遅いので，メモリアクセスステージの処理時間が他のステージに比べて長くなり，効率的なパイプライン処理を行うことはできない．ステージの処理時間を均一化するため，キャッシュの導入は必然だったといえる．キャッシュの導入により，メモリアクセスステージが1または2クロックという固定クロック数で処理できるようになった．

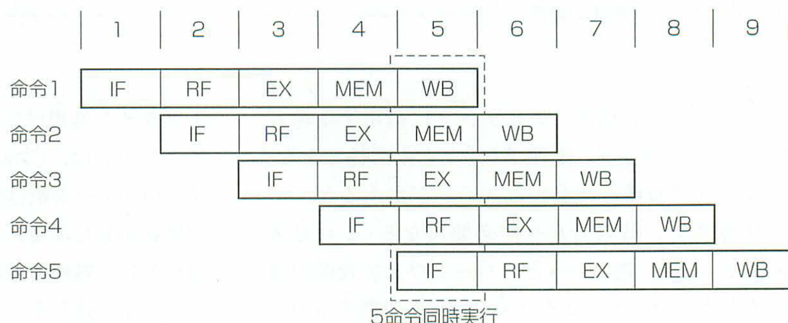
RISCのパイプラインは，コンピュータアーキテクチャの有名な教科書で学ぶことができる．それが，HennessyとPattersonによる『コンピュータアーキテクチャ』(通称ヘネパタ本)である．この教科書では，仮想的なMPUとしてDLX(デラックスと発音する)というMPUを定義し，そのパイプラインとして次の5ステージの処理が提案されている．もっともDLXはMIPSのR2000/R3000と非常に近い(同じ？)構造をしており，以下はR3000のパイプラインそのものということもできる．ただし，ヘネパタ本ではメモリがキャッシュであることをとくに強調してはいない．

● RISCのパイプライン処理

RISCのパイプライン処理を図2に示す．パイプラインがスムーズに動作する場合は，全ステージ数と同じ数の命令が(理論的には)同時実行できる．



(a) ステージと機能ブロックの関係



(b) スムーズなパイプラインの流れ

図2 RISCのパイプライン処理

(1) 命令フェッチ(IF)

命令キャッシュから命令を取り出す.

(2) 命令デコード(RF)

フェッチした命令をデコードする. 同時にレジスタオペランドをフェッチする.

(3) 命令実行(EX)

デコード結果とフェッチしたレジスタの値を基に命令を実行する. ロード/ストア命令の場合は実効アドレスの計算を行う. 分岐命令の場合は分岐先アドレスを計算する.

(4) オペランドフェッチ(MEM)

EXステージで計算したアドレスに対応するメモリの値をデータキャッシュからリードする.

(5) ライトバック(WB)

EXステージで計算した結果, またはMEMステージでフェッチしたオペランドをレジスタに格納する. ストア命令の場合はデータキャッシュにライトする.

上のパイプラインではアドレス変換のステージがないが, これはIFまたはMEMステージに先立って行われる. この詳しい説明は後半で解説するR3000のパイプラインの実際の項に譲る.

RISCのパイプラインの特徴は, アドレス計算をする専用のステージがなく, EXステージで代用している点である. このため, アドレス計算用の演算器と命令実行用の演算器(実際は加算器)をそれぞれ別個に用意する必要はない. これはRISCの「ロード/ストアアーキテクチャ」という特徴に由来する. つまり, 一つの命令では2回加算を行うことがない, 1命令で1回だけ演算器を使用するという制限の下で, レジスタとレジスタ間の加算, または, アドレス計算(ロード/ストア命令)は別の命令に分かれて定義されている.

● データハザードとフォワーディング

パイプラインの処理が乱れるハザードは, RISCのパイプラインでも発生する. それを詳しく見ていこう. まずはレジスタの依存関係に起因するハザードである. レジスタ間のリード/ライトの前後関係で, 次の4種類が考えられる.

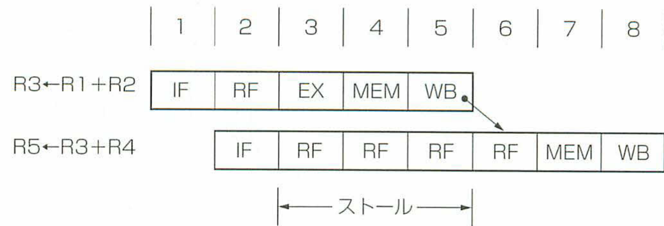
(1) RAW(Read After Write)ハザード

これは, レジスタライトの完了前に後続命令によって同一のレジスタをリードしようとした場合に生じる(図3).

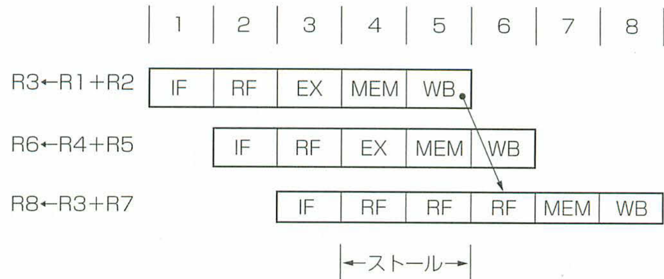
(2) WAR(Write After Read)ハザード

これは, レジスタから値をリードする前に後続命令によって同一のレジスタにライトをしようとした場合に生じる.

(3) WAW(Write After Write)ハザード



(a) 前の命令の結果(R3)を直後の命令で使用する場合



(b) 前の命令の結果(R3)を2命令後の命令で使用する場合

図3　RAW(Read After Write)ハザード

これは，同一レジスタへのライト順序が狂う場合に生じる．

(4) RAR(Read After Read)ハザード

一応挙げたが，レジスタへの変更がともなわないので，このようなハザードは存在しない．

以上はデータに起因するハザードなので，総称して**データハザード**と呼ばれる．しかし，(2)および(3)のハザードは命令の実行順序が狂わない限り発生しない．通常のパイプラインでは発生しないが，スーパースカラ構造では発生することがある．これは後半で説明する．

当面の課題は(1)のRAWハザードである．これは，フォワーディング，バイパス，または，ショートサーキットと呼ばれる手法で解決可能である．つまり，EX，MEM，WBステージからRFステージへのバイパス回路を設けることで解決できる(**図4**)．RISCでは，パイプライン処理を乱さないために，フォワーディングはなかば常識である．

しかし，パイプラインのステージ数が多い場合，具体的には，レジスタをフェッチするステージ(RF)とレジスタへの書き込みステージ(WB)の間の段数が多いと，各ステージからRFステージへのバイパス経路がその段数分必要なので，実行ステージ(EX)へ与えるデータのセレクタが巨大になってしまう．これはもちろん動作周波数にも影響を与える．どの程度フォワーディングを行うかは悩むところである．

● ロード遅延と遅延ロード

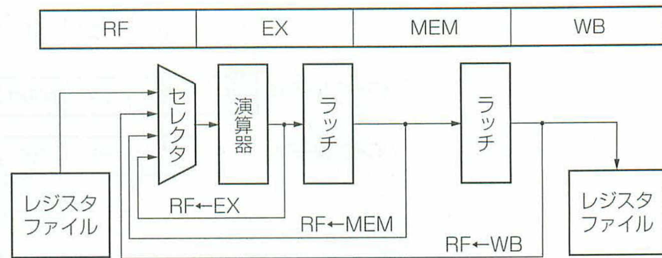
ロードした値を直後の命令で使用する場合を考える．この場合，MEMステージで値が初めて確定する．このとき後続命令はEXステージにあるのでフォワーディングは不可能である〔**図5(a)**〕．なにも対処しないと変更前のレジスタの値をフェッチしてしまう．この待ち時間を**ロード遅延**という．

このため，プログラムの意図どおりに命令を処理するには，パイプラインの**インタロック**が必要となる．インタロックとはハザードの有無をテストし，ハザードがある場合はハザード原因が解決するまでパイプラインを停止する機構である．

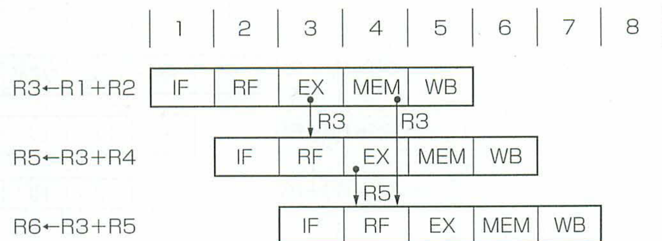
また，停止しているサイクルを**パイプラインストール**(パイプラインバブル)と呼ぶ．**図2**で示す5ステージ構成のパイプラインなら1クロックストールさせればよい〔**図5(b)**〕．

パイプラインのストールは，処理性能の低下を意味する．それを回避する手法の一つは，プログラムの意図を損わない範囲で命令の順序を入れ替えることである．いまの場合，1クロック分(1命令分)待ち合わせればいいので，ロードした値を参照する命令と後続の無関係な命令を入れ替えればよい．

入れ替えるべき適当な命令がない場合は，NOP命令を挿入することになる〔**図5(c)**〕．この手法はデータハザードの回避にも有効である．このような命令入れ替えや命令挿入を，**命令スケジュール**と呼ぶ．



(a) バイパス回路

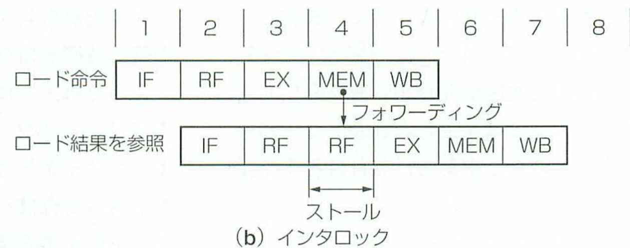


(b) フォワーディング（ストールしない）

図4　バイパス回路とフォワーディング



(a) 遅延ロード



(b) インタロック



(c) NOP命令を挿入

図5　遅延ロード

RISCのアセンブラは命令スケジュールを当然のように行っている(禁止の設定も可能)．つまり，アセンブラが「勝手に」最適化するので，プログラマが書いたとおりの順序でコード生成が行われるとは限らないのである．この事実を知ったとき，筆者は少々衝撃を受けたが，いまでは慣れてしまった．

RISCは制御構造の単純化を目標としているから，インタロックは歓迎すべきものではない．ロード遅延をそのまま許し，アセンブラによる命令スケジュールによってのみストールを回避しようという考えが



図6　遅延分岐

ある．これが**遅延ロード**である．MIPSのR2000/R3000は遅延ロードを許すアーキテクチャを採用している．

ただし，R4000からはインタロックするアーキテクチャに変更された．これは，現実問題として，命令の並び替えができる場合が少なく，多くの場合はNOP命令が挿入されてしまうからであろう．NOP命令の挿入により，全体としての命令処理は1クロック余分にかかるが，これはストールで1クロックインタロックしても同じである．それならNOP命令がない分，命令コードのサイズを小さくできるという利点がある．

● 制御ハザード

パイプライン処理を乱すハザードにはデータハザードのほかに**制御ハザード**がある．これは分岐によるハザードである．ブランチハザードともいう．RISCでは，条件分岐は汎用レジスタの値で分岐条件を決定する．MPUによっては，CISCと同じく条件フラグを採用しているものもある．この場合の制御ハザードはフラグハザードともいう．

さて，条件分岐の場合，分岐条件が確定するまで分岐先の命令フェッチができない〔**図6**(**a**)〕．これによるストールは命令スケジュール(条件確定を早くする)で回避できる場合もある．

条件フラグを使用する場合でも命令スケジュール可能だが，そのアルゴリズムは非常に難しい．RISCが条件フラグを採用しない理由の一つは，コンパイラでの命令スケジュールを簡単にするためである．なお，条件分岐で分岐条件が成立して分岐することをTAKEN，分岐条件が成立せず分岐しないことをNOT TAKEN(あるいはNO TAKEN)という．



図7　命令デコードが2ステージの場合の制御ハザード

● 遅延分岐

パイプライン処理を乱さないため，ストール期間中も（通常は無効化してしまう）分岐命令の後続命令（これを遅延スロットという）を実行させるという考えがある．**図2**に示すパイプラインでは，EXステージでTAKEN/NOT TAKENが決定される．したがって分岐先の命令フェッチは，1クロックのストール後に実行可能である．

TAKENする場合，通常なら分岐命令の後続命令は実行を禁止しなければならない．しかし，その遅延スロットの命令を実行してから，分岐先の命令をフェッチする構造にすれば，パイプラインはストールする必要はない〔**図6（b）**〕．TAKENしない場合は，もともとストールしない．これを**遅延分岐**と呼ぶ．

このような遅延スロットを設ければ，命令スケジュールを行うことができる．分岐命令の前方にある命令を遅延スロットにもってくることで，分岐命令によるストールはなくなる．ただし，遅延スロットに入れる適当な命令がない場合は，NOP命令を入れることになる．

R2000/R3000のパイプラインはこのようになっているが，現実問題としては，分岐命令の分岐先アドレスもEXステージで計算される（したがって，分岐条件を判断するための専用の演算器が別個に必要である）ため，それとほぼ同時に分岐先を命令フェッチするのはタイミング的に厳しい．動作周波数を向上させるためには，遅延分岐を採用しつつも，もう1クロック遅れさせるのが望ましい〔**図6（c）**〕．このあたりをうまく回避するのが回路設計技術であるということもできるが…．一般的には，分岐予測を行うことでストールを解消することが可能である．

さて，制御ハザードではTAKENの決定が遅いほどストール期間が長くなる．これはステージ数の多いパイプラインで顕著になる．たとえば，可変長命令を採用するx86のようなMPUにおいては命令デコードに時間がかかる．一般的には，パイプラインで少なくとも2ステージ分が必要である．

たとえば，

IF　RF1　RF2　EX　MEM　WB

の6ステージからなるパイプラインを考える．TAKENの決定はEXステージなので，これまでの説明より1クロック遅いことになる．このとき分岐命令でのストールは2クロックである（**図7**）．1クロックを遅延スロットで埋め合わせるとしても，さらに1クロックだけ処理に余計な時間がかかる．あとで述べるスーパーパイプラインでは，EXステージより前のステージ数がさらに増加し，分岐命令のストールによる性能低下は深刻なものとなる．

● 分岐予測

分岐命令の処理を高速化するために，**分岐予測**という機構が採用される．これは，分岐先アドレスをパイプラインのより早いステージで生成し，分岐先の命令フェッチを早期に行う手法である．具体的には，分岐ターゲットバッファ（BTB：Branch Target Buffer），または分岐予測テーブル（BPT：Branch Prediction Table，BHT：Branch History Table）と呼ばれるキャッシュを用意し，分岐命令のアドレス，分岐履歴情報，予測される分岐先アドレスを格納しておく．

命令フェッチ時（IFステージ）にBTBを参照し，ヒット（登録してある分岐命令のアドレスと命令フェッチアドレスが一致）すれば，分岐履歴情報にしたがって，分岐先アドレスを出力し，命令フェッチを行いながら，TAKEN/NOT TAKENの判定を待つ．予測が成功すればフェッチした命令をそのままデコードすればよい．

予測が失敗すれば，実際にEXステージで計算されるアドレスから命令フェッチをやり直し，BTBの分岐履歴情報を更新する（**図8**）．BTBにヒットし予測が成功する場合はストールがなくなる．BTBにヒットしない場合は，分岐予測を行わない場合と同じタイミ

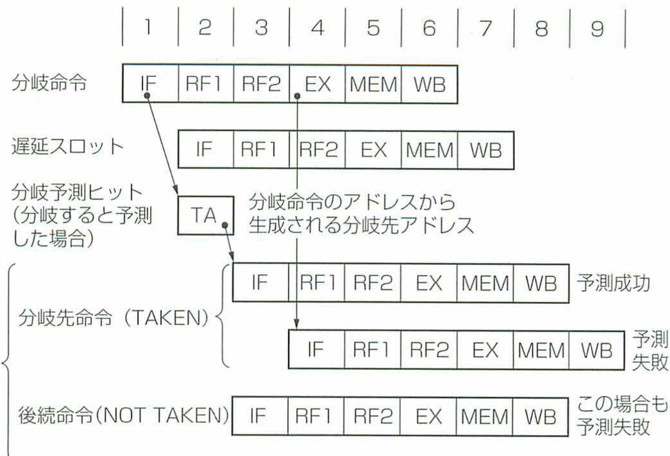


図8　分岐予測

ングで分岐命令が処理される．

しかし，BTBにヒットするのに予測が失敗する場合は，何もしない場合に比べて，パイプラインの回復処理にかえって時間がかかってしまうことがある．これが，分岐予測失敗時のペナルティである．したがって，分岐予測を採用しても予測が失敗ばかりすると，かえって性能が低下するのでヒット率を向上させるための工夫が必要である．

図8のパイプラインのモデルではBTBにヒットすると予測した分岐先アドレスから命令フェッチを行うが，MPUによっては(予測する)分岐先の数命令をBTBに格納しておき，そこから命令をフェッチする方法を採用する．こうすることにより，パイプラインは予測していない方向の命令も同時にフェッチできるので，分岐予測が失敗した場合のペナルティを最小化できる．

また，分岐予測の成功する確率が高いと思われる場合は，TAKEN/NOT TAKENが決定するまで，予測した分岐先から命令をどんどん先取り(プリフェッチ)する手法もある．パイプラインのステージ数が大きく，TAKEN/NOT TAKENの決定がパイプラインの遅い(後段の)ステージで行われる場合，予測が成功すれば効果的である．逆に予測が失敗したときのペナルティは大きくなる．分岐予測の成功率によほどの自信があるか，失敗時の回復処理がかなり高速化されてないと採れない方式であるが，最近のMPUではけっこうポピュラーである．

● 分岐予測の方法

予測の方法は分岐履歴情報による場合が多い．これは分岐する確率を示す1～2ビットのフラグであり，BTBに登録されている分岐命令ごとに存在する．分岐履歴情報が1ビットの場合は1であるとき「分岐する」，0であるとき「分岐しない」と予測する．これは，その分岐命令が過去1回で分岐したか否かを示している．つまり，以前分岐した分岐命令は今回も分岐すると予測するわけである．

分岐履歴情報が2ビットの場合はもう少し慎重である．ビット列への意味のもたせ方はいろいろ考えられるが，たとえば，11，10で「分岐する」，01，00で「分岐しない」と予測する．これは，その分岐命令が，過去2回において何回「連続して」分岐したかを示す．分岐する傾向が大きい方向に予測するわけだ．

なお，分岐する(と予測する)分岐命令のみをBTBに登録する方法もある．この場合は分岐履歴情報は不要で，BTBにヒットすれば「分岐する」，ヒットしなければ「分岐しない」と予測する．この場合，分岐予測が成功する確率は，分岐履歴情報が1ビットの場合とほぼ同等であるが，BTBの回路規模は約半分になる．

分岐予測を行わない場合で，分岐命令を高速化する方法として，分岐先と分岐元の命令を同時にプリフェッチする手法もある．これに関係する特許は，昔は山のようにあった．この方法は回路規模が大きくなるため，あまり現実的でない．といいつつも，Intel系のMPU(とくにIA-64)ではそのような説明をよく目にする．ただし，具体的な実装方法は不明である．米国の特許をよく調べればわかるかもしれない．

● 構造ハザード

構造ハザードとは，パイプラインの二つ以上のステージが一つしかないハードウェア資源を取り合うた

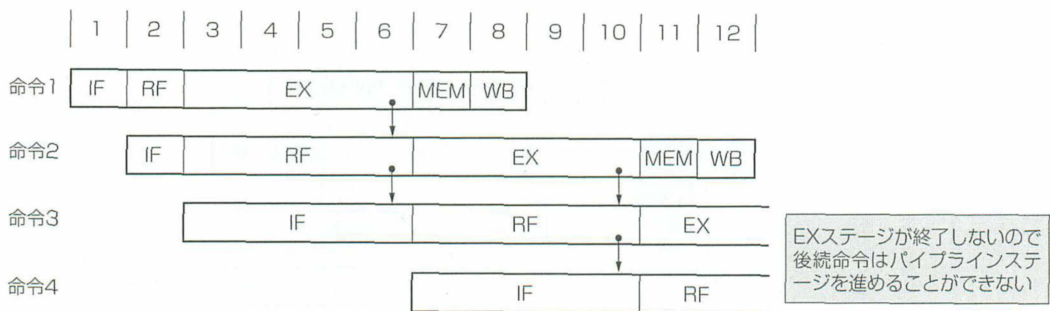


図9　実行ステージが長いパイプライン

めに生じるハザードである．たとえば，5ステージで構成されるパイプラインでは，1時刻に五つすべてのステージが実行される可能性がある．もし，各ステージで同一の演算器などを使用する場合は競合するので，優先されるステージ以外は待ち合わせをする必要がある．

RISCの場合，ほとんどのハードウェア資源は競合しないように設計されているのであまり問題はない．しかし，例外もある．それはキャッシュ（あるいはメモリ）である．**図2(b)**をもう一度見ていただきたい．時刻4において命令1のMEMステージと命令4のIFステージが重なっている．もし，命令1がロード/ストア命令であり，命令とデータキャッシュの区別がなく単一のキャッシュしかない場合は，IFステージもMEMステージもキャッシュアクセスなので，資源の競合が生じる．キャッシュが存在しない場合もメモリアクセスの競合が生じる．この場合は，先にある命令1のMEMステージを優先させ，命令4のIFステージをインタロックして待ち合わせることになる．これは，できるだけパイプラインをインタロックさせないというRISCの考え方に反する．

そこで，多くのMPUでは命令とデータを二つのキャッシュに分割して同時にアクセスできるようにしている．これならアクセスの競合によるインタロックは発生しない．このように命令とデータの供給経路を独立させる方式をハーバードアーキテクチャという．

なお，命令とデータに関しては，TLBが一つしかない場合，アドレス変換時にも資源の競合が生じる．それを避けるため，命令用とデータ用のTLBを独立に用意するアーキテクチャもある．多くの場合，命令はアクセスするアドレス範囲が小さい（あるいは連続している）ため，命令用のTLBをマイクロTLBとして，仮想アドレスと物理アドレスのペアを本当のTLBからキャッシュして持っているのが普通である．

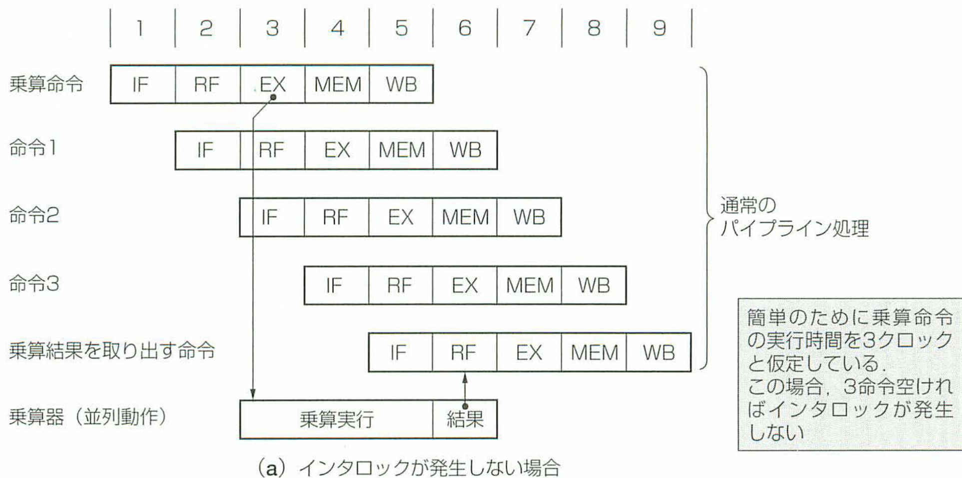
● ステージの処理時間が不均一なパイプライン

さて，パイプラインのステージ間の実行時間が均一でない場合を考える．RISCは命令を1クロックで実行するのが基本であるが，乗除算や浮動小数点演算など1クロックで実行するのが難しい場合もある．

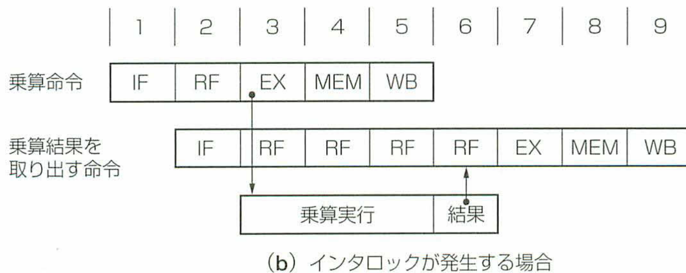
いま，実行ステージ(EX)の処理が4単位時間かかるものとする(**図9**)．この場合，EXステージが終了するまで同時実行中の他のステージも待ち合わせをするので，パイプラインのスループットは実行ステージの処理時間に依存する．ほかのステージの処理時間は実行ステージの処理時間に隠れてしまう．実行ステージの処理時間が長いだけならまだよい．ほかのステージもまちまちな処理時間を有する場合はもっと悲惨である．不均一であればあるほど，パイプラインの処理時間は各パイプラインステージの処理時間の総和に近づいていく（パイプラインの意味がなくなる）．このため，実行ステージ以外のステージの処理時間を均一にすることが肝要である．

パイプラインにおいて実行（処理）時間がかかるのは，特定命令の「実行ステージ」のほかに，メモリの速度に依存する「命令フェッチステージ」や「オペランドフェッチステージ」がある．RISCは，キャッシュを採用することで命令フェッチやオペランドフェッチの処理時間を1クロックに押し込めようとしている．

典型的なRISCであるMIPSアーキテクチャにおいては，全命令の実行クロックを1クロックとするために，実行時間がかかる乗除算は，通常のパイプラインとは独立して並列実行する．そして，乗除算の結果は汎用レジスタではなく，専用のレジスタに格納される．つまり，乗除算命令では汎用レジスタ間のデータハザードは発生しない．このため，乗除算命令の処理は通常のパイプライン動作に影響を与えない．乗除算が完了した後で，専用レジスタから演算結果を取り出せば（専用レジスタから汎用レジスタへの転送命令が用意



(a) インタロックが発生しない場合



(b) インタロックが発生する場合

図10　MIPSの乗除算命令

されている）インタロックは発生しない．

初期のMIPSプロセッサであるR3000では，乗算と除算の実行時間がそれぞれ12クロックと35クロックである．乗算命令に関していえば，実行を開始してから12クロック後に結果を取り出せばインタロックは発生しない〔**図10(a)**〕．プログラム的には乗算命令と結果を取り出す命令の間が12命令分空いていればよい．

一方，12クロック未満で結果を取り出そうとすると，アーキテクチャ的には不本意ながらインタロックしてしまう〔**図10(b)**〕．現実的には乗除算命令と結果を取り出す命令の間はせいぜい3命令程度しか空けることができないので，乗除算命令があるとほとんどの場合インタロックしてしまうのだが，コンパイラの頑張りによってはインタロックしない可能性を残している．

3　パイプラインを効率良く動かす各種の方法

● 効率的なパイプライン処理が可能になった理由

歴史的に見れば，キャッシュメモリ（高速なローカルメモリ）がまだ高価で外付けのキャッシュすら現実的でなかった時代，プロセッサの処理はメモリからの命令フェッチにいちばん時間がかかっていた．当然の流れとして，プロセッサの性能を上げるためには，フェッチする命令数を減らすこと，1命令で行う処理を増やすことが考えられた．結果として，上述したように実行ステージが長くなる傾向になるのだが，多くの場合はいちばん時間のかかる命令フェッチと，あまり時間のかからない命令のデコードおよび実行をオーバラップ（パイプライン処理）させて実行効率を上げることが可能になる．これが，その時代の最適解であった．そして，これこそがCISCの考え方である．

その後RISCという選択肢が現れてきた背景には，キャッシュが一般的になり，命令フェッチがもはやプログラムの実行に支配的でなくなったことがある．命令のデコードや実行時間が命令フェッチ時間の影に隠れなくなり，実行する命令数よりも1命令の実行時間のほうが性能に対し支配的になった．RISCでは，基本的に1クロック実行なので，CISCに比べて命令実行時間が1/3から1/5になる．1命令が単純な分，同じ処理に要するコード量は増加するが，RISCになる

表1　x86系CPUのCPIとMIPS値

CPU	CPI
8086	15.0
80286	6.0
80386	4.5
i486	1.7

(a) x86系CPUのCPI

CPU	MHz	MIPS
8086	5	0.33
80286	8	1.33
80386	12	2.67
i486	25	14.71
Pentium	66	110

(b) x86系CPUのMIPS値

ことによる命令数の増加はわずか30～50％であるというから，これを差し引いても性能は向上する．

また，RISCでは命令が基本操作に限定されているので，コンパイラによる最適化が行いやすいという利点もある．まあ，現実には，基本的な命令だけで優れた最適化ができるということをMIPSやSPARC用のコンパイラが実証できたためにRISCがメジャーになったともいえるのだが，CISCからRISCの流れは歴史の必然でもあった．

● CPIとMIPS値

パイプライン処理における命令の実行効率を表す指標として，CPI(Clock cycles Per Instruction)がある．これは1命令を実行するのに必要なクロック数である．RISCの当初の目標は，キャッシュと効率的なパイプライン処理でCPIを1にすることにある．実際，RISCはキャッシュにヒットしパイプラインにインタロックがない場合はCPIが限りなく1に近づく．IntelのMPUの平均CPIに関しては，80386やi486を設計した技術者の一人であるPatrick Gelsingerのレポートがある．それによると，**表1(a)**のような値が出ている．

MPUが進化するにつれてパイプラインの効率が上昇しているのがわかる．さすがインテルというところだろうか．i486でCPIが急激に改善したのは，キャッシュの恩恵といわれている．CISCでありながらRISC並みのパイプライン処理を採用したことも一因であろう．現在のPentiumのCPIは0.6～0.7であるという(ちょっと性能が良すぎる感もあるが)．これは次章で説明するスーパースカラの恩恵である．

CPIはMIPS(Million Instructions Per Second)値と密接な関係がある．MIPS値とは1秒間に実行できる命令数(100万命令単位)だから，動作周波数とCPIが決まれば，

周波数(MHz単位)÷CPI

という計算式で，MIPS値が求まる．この式で，上のx86プロセッサのMIPS値を計算すると**表1(b)**のようになる．

実際に公表されるMIPS値は，Dhrystone MIPS(最近ではDMIPSと略記されることもある)なので，もう少し高い値になっているかもしれない．これは，Dhrystoneベンチマークを実行した性能が，1MIPS相当のVAX-11/780の何倍であるかを表すものである．

Dhrystone MIPSでは，コンパイラの性能しだいでシングルパイプラインのMPUのCPIが1を割ることも多く，直感的ではない．しかし，現在実際に使用されているMIPS値はDhrystone MIPSが主流なので，慣れが必要である．

もっとも最近のx86系は，MIPS値の公表をやめてしまっている(表向きの理由はいろいろあるが，発表するとCPIの大きさが問題となるからだろう)ので，性能を比較するためには動作周波数に頼るしかない．各メーカーは独自の基準で従来品との相対性能を公表しているが，異なるメーカー間での性能比較はできない．いくら動作周波数が高くてもCPIが悪ければ何にもならないのだが，メーカーやマスコミはこの点を意図的にうやむやにしているようにも思える．

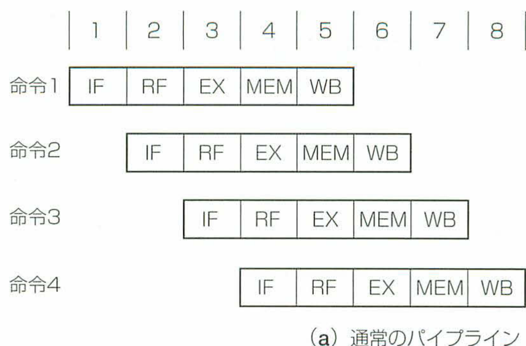
IntelはPentium4で3GHz以上の動作周波数を実現した．実効性能はPentium4と同等であるが，動作周波数ではPentium4に劣るAthlonXPを有するAMDは，周波数の大きさによる優位性のアピールから実効性能の優位性のアピール(モデル番号の採用)に方針転換した．

● スーパーパイプライン

MPUを高い周波数で動作させるためには，パイプラインの1ステージあたりで実行する論理を減少させる必要がある．単純に考えると従来1ステージで実行していた処理を2ステージに分割することである．つまり，高速な動作周波数になるにつれてパイプラインのステージ数が増加する傾向にある．いま，パイプラインのステージを，

IF1　IF2　RF1　RF2　EX1　EX2　MEM1　MEM2　WB1　WB2

としてCPIを試算してみよう．**図11(a)**では4命令を8クロックで実行しているのでCPIは2.0である．一方，**図11(b)**では4命令を13クロックで実行しているのでCPIは3.25である．スーパーパイプライン構成にする



(a) 通常のパイプライン



(b) スーパーパイプライン

図11
スーパーパイプラインの効果

ことでCPIは約1.5倍に増加してしまう．しかし，動作周波数を2倍に引き上げることができれば実質的な性能は向上する．これがスーパーパイプラインの考え方である．

スーパーパイプラインを最初に採用したのはMIPSのR4000である．これは当初100MHz動作であったが，最終的には250MHz動作を達成している．ほぼ同時期に登場したDECのAlpha(21064)は200MHz動作を達成していた．これは1990年代の初めとしては驚異的な動作周波数だった．このため，Alphaは世界最高速のMPUとしてギネスブックに登録された．

最近では，

動作周波数を上げる＝パイプラインのステージ数を増やす

という図式が常識のように語られるようになった．IntelのPentium4(Willamette)は20ステージのパイプライン構成で2GHz以上の動作を目指した．そして，Northwoodで3GHzを超えた．IPコアの分野でも，Lexra社がLX4189(MIPS系)でパイプラインを従来の5ステージから6ステージに変更することで，初めて250MHz以上の動作周波数を達成したと発表した．

動作周波数を上げるためにはパイプラインのステージ数を増やす必要があるのは本当だが，逆は必ずしも真ならずなので，そんな単純なものではない．しかし，これからのMPU設計においては，パイプラインのステージ数を増加して動作周波数をかせぎ，それによるCPIの増加は分岐予測を高度にすることで補っていく傾向になるのは間違いないだろう．

● プリフェッチとデカップル(decouple)構成

命令フェッチが命令キャッシュにヒットする限りは，各サイクルごとに命令デコーダに命令が滞りなく供給されるので，プリフェッチして命令をFIFOなどに蓄えておく必要はない．しかし，命令キャッシュミスが発生すると命令供給が停止するので，パイプラインがストールしてしまう．それを防ぐためにプリフェッチは有効である．命令デコード以降のパイプライン処理とは，命令を絶えず独立にプリフェッチしておけば命令デコードにおいても命令の供給が停止する頻度は少なくなる．

命令キャッシュのミスが発生した場合，命令キャッシュへの書き込みと同時にデコーダへ命令をバイパスする「命令ストリーミング」もパイプラインのストールを低減させる方法の一つである．しかし，命令ストリーミングでは，(通常は)パイプラインクロックよりも遅いバスクロックに同期して命令供給が行われるので，命令ストリーミング中の命令処理はバスクロック

Column ウェーブパイプライン

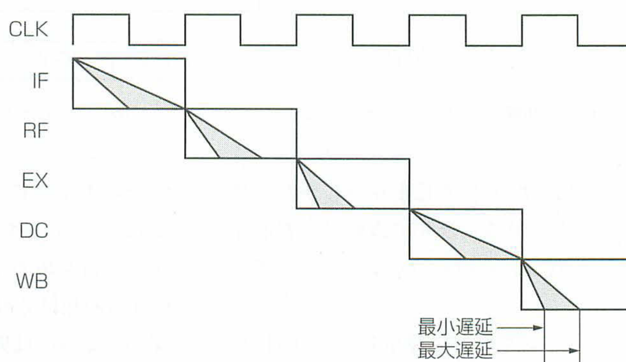
パイプライン処理は，各ステージの処理を，通常，一つのクロックに同期させて進めていく．しかし，各ステージの純粋な処理時間は論理の複雑さに依存し，クロックで既定される時間ぎりぎりまでかかる場合もあれば，クロックで既定される時間より早く終わる場合もある．

パイプラインのあるステージがクロックのサイクルより短い時間で終了する場合，その空き時間をむだにしないような実装ができれば，相対的に処理時間を短縮することができ，見かけ上のクロック周波数を向上させることができる．

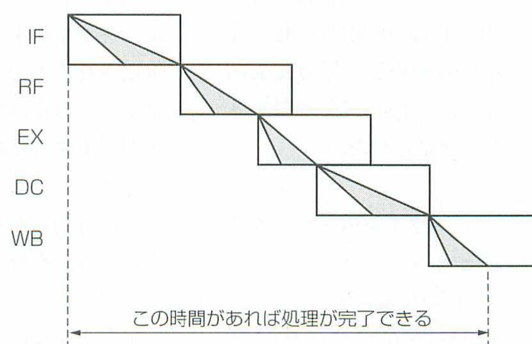
図Aを見てほしい．IF，RF，EX，DC，WBからなる5ステージのパイプラインを考える（MEMステージはデータキャッシュアクセスなのでDCとしている）．図に示すように，各ステージの処理時間を仮定する．ここでは，IFとDCがキャッシュアクセスで，もっとも遅いステージになっている．RFがデコードステージでその次に遅い．EXとWBはレジスタアクセスなので比較的高速である．

従来のパイプライン方式では，**図A(a)**のように5クロックかけて1命令の処理が終了する．このステージの空き時間を詰めていくと，**図A(b)**のように，4クロック程度で1命令の処理が終了する．処理時間が4/5になったのだから，同じ処理をする場合のスピードは1.25倍になる．つまり，見かけ上のクロック周波数は1.25倍になる．

しかし，MPUの動作はクロック同期が基本であるが，各ステージを駆動するクロックの周期（周波数）さえ一致していれば，各ステージを同じタイミングで処理しなくても，安定なパイプライン処理を維持することができる．単純には，ステージごとに独立なクロックを用意することが考えられる．**図B**にパイプラインのステージ数と同じ5相クロックを用いたパイプラインを示す．



(a) 単相クロック同期



(b) クロックにしばられない場合

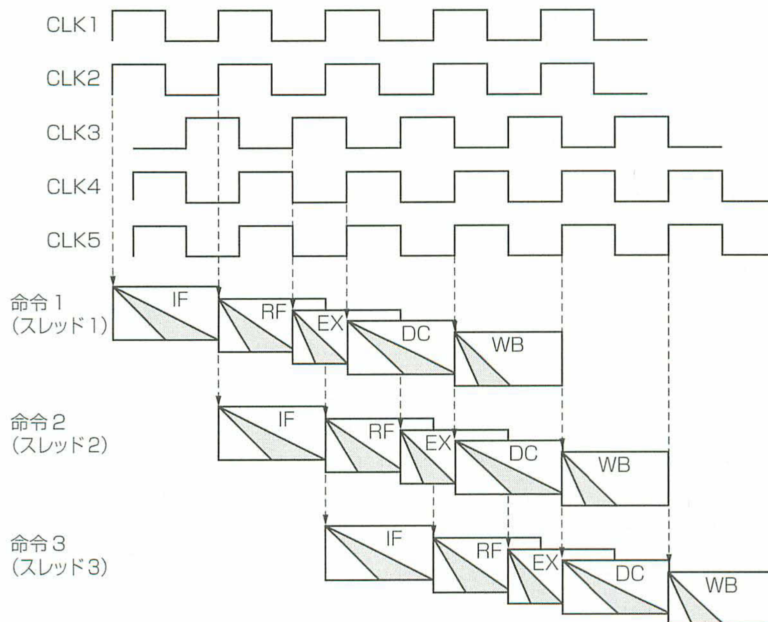
図A　ウェーブパイプライン

この場合，5系統のクロックの周波数は同一であり，IFステージはCLK1，RFステージはCLK2，EXステージはCLK3，DCステージはCLK4，WBステージはCLK5に同期して動作している．このように多相クロックを用いることで，見かけ上のクロック周波数を向上させることができる．このようなパイプライン構造をウェーブパイプライン（wave pipeline＝波打ったパイプライン）と呼ぶ．あるいは，最大レートパイプライン（maximum rate pipeline）と呼ばれ，文字どおり，パイプラインの処理速度を最大限に上げることができる．

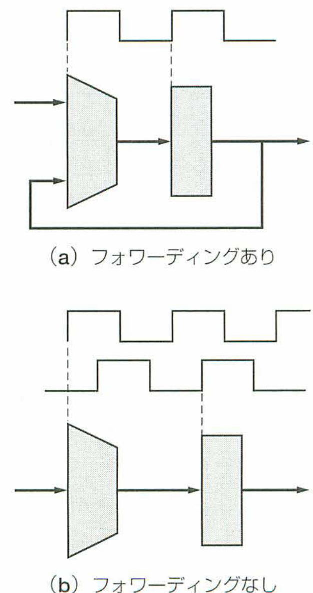
ウェーブパイプラインを行うためにはいくつか制限がある．各ステージのクロックが一致していない（ずれがある）ので，WBステージの前の結果をRFステージにフォワーディングすることが難しい（**図C**）．フォワーディングができなくなると，見かけ上の動作周波数は向上しても，ハザードによりCPIが増加してしまう．ウェーブパイプラインは，ハザードが発生しにくい状況下でこそ効果を発揮する．

ハザードが発生しにくい場合とは，どういう状況であろうか．これは，パイプラインに順次投入されていく命令間に依存性のない場合である．この例としてすぐに思い付くのは，多数のスレッドをパイプライン処理する場合である．スレッドとはプログラムのうちで並列実行できる部分を抽出したもの（ゆえに，多くの場合，依存性はない）である．たとえば，近年はやりのマルチメディアアプリケーションはスレッドに分解するのに適している．このため，ウェーブパイプラインは多くの場合，マルチスレッドの処理に適用される．

ウェーブパイプラインは，大学レベルでは多くの研究がなされているが，商用のものはまだ存在しない．これは多相クロックを使用するため，クロックの遅延を合わせ込むのが難しく，遅延解析に向かないためである．つまり，安定量産できるための回路設計が難しいのである．さらに，多相クロックの場合，論理合成が難しく，手作業による専用回路設計が必要になる．それにかける工数に比して利益の見込みが少ないので，企業が実践するには苦しいものがある．



図B　5相クロックを用いたウェーブパイプライン



図C　演算後のフォワーディング

同期に近くなり，効率があまりよくない．プリフェッチは，命令キャッシュミスの発生が契機となるわけではなく，無条件に命令フェッチを行っていくので，命令ストリーミングよりも効率がよい(はずである)．

シングルパイプラインではあまりお目にかからないが，**デカップル方式**という構成がある．これは，プリフェッチとよく似た概念であるが，命令デコードと実行ステージの中間にFIFOを置いて，そのFIFOに絶えずデコード済みの命令を格納しておく．こうすることで，ソースオペランドが有効である限り，FIFO内の命令は各サイクルごとに命令実行を開始することが可能になる．つまり，オペランドの依存性による命令デコードステージでのストールが緩衝されて見えなくなる(パイプライン効率が上がる)．

当然，命令フェッチとデコードまでのステージと実行ステージ以降は別のクロックに同期し，独立して動く．パイプラインがデコードまでと実行以降に分離(decouple)されていることで，デカップル方式と呼ばれている．

デカップル方式の利点は，単純なプリフェッチとは異なり，命令デコードを行うので分岐命令を認識することが可能であること，そして分岐予測をしながら投機的(speculative)に命令のプリフェッチを行うことができる点である．単なるプリフェッチであれば，分岐する分岐命令以降にある命令をむだにプリフェッチするおそれがある．分岐予測にしたがってプリフェッチを行うことができれば，(分岐予測が当たる限り)命令フェッチのロスはなくなる．

このため，デカップル方式では，分岐予測が有効に働けば，パイプライン処理の中で，命令フェッチと命令デコードステージを無視することができる．たとえば，5ステージのパイプライン処理ならば，2ステージ少ない，3ステージのパイプラインと同等の効率で命令を処理できる．

プリフェッチやデカップル方式での投機的なデコードは，実行ステージ以降で発生するパイプラインストールの合間を縫って行われる．実行ステージ以降にストールがまったく発生しなければ，プリフェッチ機構自身が無意味なものになってしまう．パイプライン効率は落ちないが，プリフェッチをしてもしなくても同じ効率にしかならないので，余分な回路ということになる．

実際問題として，シングルパイプラインではロード遅延とデータキャッシュミス以外では実行ステージ以降でのストールは発生せず，プリフェッチは，その回路規模の割には，性能は向上しないと思われる．しかし，2命令以上を同時に処理するスーパースカラにおいては，命令デコードの倍以上の速度で命令が処理されていくので，プリフェッチや投機的デコードの機構を用意しておかなければ命令供給が命令消費に追いつかなくなる．デカップル構造についてはスーパースカラの解説の章(第3章)で詳しく説明する．

● 自己書き換えとパイプライン

昔，8086や68000というMPUが全盛だった頃，プログラムのコードサイズを削減するために，命令コード領域をストア命令で書き換えて実行するという技が重宝されていた．これは**自己書き換え**と呼ばれる手法である．

自己書き換えは，パイプラインを採用するMPUでは期待どおりの動作をするとは限らない．それは，パイプラインのステージを考えれば明らかで，書き換えた命令のフェッチ(IF)は書き換える命令のライト(WB)以降でなければならないためである．たとえば，

IF　RF　EX　MEM　WB

という5ステージ構成では，最低5命令以後を書き換えなければ，そこを正しくフェッチできない(**図12**)．また，命令のプリフェッチを行う場合に，一概に何命令後を書き換えれば大丈夫かということは保証できない．書き換えた場所にジャンプしさえすればよいという考えもある．この方法も，分岐予測などで命令フェッチが先行する場合は，うまくいかないことがある．

ところで最近のMPUは，命令キャッシュとデータキャッシュが分離されているので，単純に命令コードを書き換えることはできない．ストア命令を実行してもデータキャッシュの内容が変更されるだけで，命令キャッシュの内容は変わらないからである．

ただし，(OSに限られるが)特権命令を使えば，書き換えたアドレスに対応する命令キャッシュの内容を無効化することで，自己書き換えを実現できる．もし，ライトバックキャッシュ構成ならデータキャッシュを最初に強制的にライトバックさせることも必要である．

……と，自己書き換えを推奨するような説明をしてみたが，最近のプログラミングではこの技法は好ましくないとされている．現在はMMUが内蔵され，十分大きなアドレス空間を使ってプログラムを作ることも可能なので，わざわざプログラムの流れを分かりにくくする自己書き換えを行う理由はない．

とはいえ，仮想記憶のデマンドページングで行われ

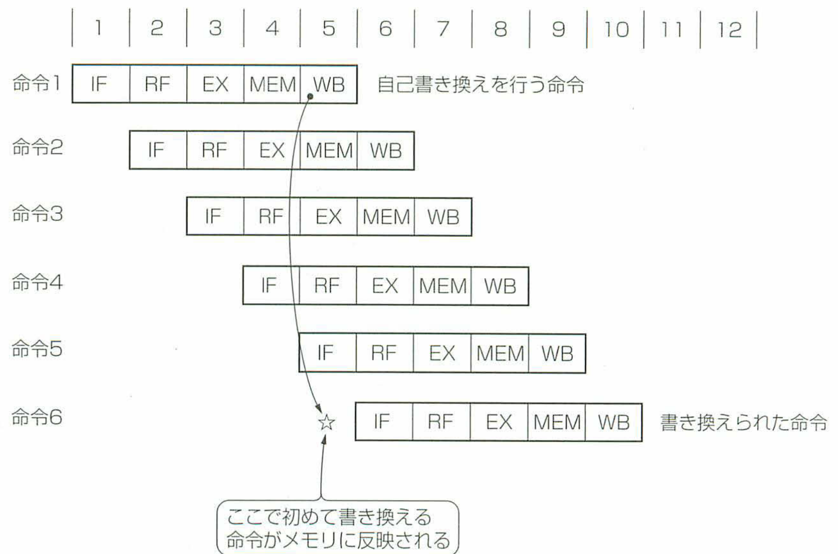


図12
命令書き換えのタイミング

るスワップインは壮大な自己書き換えではないかと考えると，OSなら自己書き換えをしてもいいのかという話も出てくる．リアルタイムOSなどのプログラムのダイナミックリンクも，自己書き換えに近い．

もっとも，有限オートマトンとしてのコンピュータを考えれば，自己書き換えができるのは当然の機能/属性である．プログラムで行う自己書き換えとOSのページングは粒度(書き換えから実行までの時間的空間的距離)の大きさの違いとして説明される．つまり，粒度の小さい自己書き換えは推奨されないということだ．

パイプライン処理の概念

4 R3000のパイプライン

● RISCの基本そのままのパイプライン

R3000のパイプラインは，基本的には前半の**図2**で示したRISCのパイプラインと同じである．IF，RF，EX，MEM，WBの5ステージで構成される．実際には$\phi 1$，$\phi 2$の2相クロックで動作し，1クロック間に2ステップの処理を行っている．**図13**にR3000のパイプラインの詳細を示す．各ステージでの動作は，次のようになっている．

1) IF $\phi 1$

マイクロTLB(ITLB)を使用して命令の仮想アドレス(IVA)を物理アドレスに変換する．分岐先アドレスはRFステージの$\phi 2$で計算され，EXステージの$\phi 1$でアドレス変換される．

2) IF $\phi 2$

物理アドレスを命令キャッシュに転送し，命令キャッシュをアクセスする(ICache)．

3) RF $\phi 1$

命令キャッシュのヒット/ミスがチェックされ，命令キャッシュから命令を読み出す(ICache)．

4) RF $\phi 2$

命令をデコードする(ID)．分岐命令の場合は分岐先アドレスを計算する．レジスタファイルをリードする(RF)．

5) EX $\phi 1 + \phi 2$

オペランドを他のパイプラインステージからバイパスし，演算する(ALU)．ストアするデータがあれば位置合わせを行う．

6) EX $\phi 1$

分岐命令ならTAKEN/NOT TAKENを決定する．ロード/ストア命令ならオペランドの仮想アドレスを計算する(DVA)．

7) EX $\phi 2$

ロード/ストア命令ならTLBを使用してオペランド

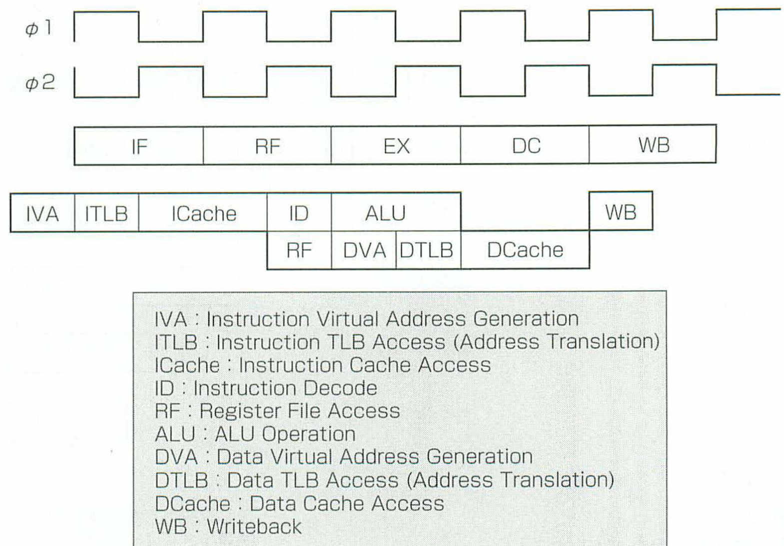


図13　R3000のパイプライン

の仮想アドレスを物理アドレスに変換する(DTLB).

8) MEM φ1

ロード/ストア命令なら物理アドレスをデータキャッシュに転送し，データキャッシュをアクセスする(DCache).

9) MEM φ2

データキャッシュのヒット/ミスがチェックされ，命令キャッシュからオペランドを読み出す(DCache).

10) WB φ1

EXステージでの演算結果をレジスタファイルに書き込む(WB).ストア命令の場合はデータキャッシュに書き込む.

● 単相クロック動作のR3000

R3000相当のIPコアを提供する目的で後年に発表された，パイプラインを見直した4Kc(Jade)，4KEc(Emerald)，5Kc(Opal)など(これらはコアの名称)は，単相クロック同期に変更しているが，基本的なパイプライン構造に変更はない．**図14**にR3000とJade，そしてOpalのパイプラインの比較を示す．R3000のパイプラインが**図13**と一部異なっているが，**図13**は説明用に簡略化したもので，**図14**は現実に近いものと理解すればいいだろう．

図14に示すように，単相クロックでの再設計を考えた場合，R3000は多くのクリティカルな操作(命令キャッシュアクセス，レジスタリード，データTLB参照)をパイプラインクロックの立ち下がりエッジに同期して行っている．また，データキャッシュアクセスはクロックの立ち上がり同期であり，データキャッシュからリードしたデータの位置合わせ(**図14**のLA)を同じパイプラインステージ内で行うので，タイミングはかなり厳しい．命令のアドレス計算も，命令キャッシュアクセスの前後に，二つの1/2サイクルのアクセス(IA1，IA2)に分割して行われるので，制御が複雑になる．これらが，IPコアとして容易に論理合成を行うためのボトルネックになっている．また，SRAM(キャッシュ)のアクセスタイミングも厳しく，キャッシュをメモリコンパイラなどで自動生成するのが困難である．

このため，Jadeではパイプラインが再設計された．具体的には，すべての操作を1フェーズ早めてクロックの立ち上がり同期にした．さらに，命令TLBアクセスとデータキャッシュアクセスを1ステージ早くして，リードデータの位置合わせをキャッシュアクセスと別のステージにもっていった．結果として，すべてのクリティカルな操作は立ち上がりエッジ同期になった．命令のアドレス計算は，命令キャッシュアクセス後の，一つのパイプラインステージに統合された．これらの結果，データキャッシュアクセスのタイミングに余裕ができた．

図14からわかるように，レジスタファイルへのライトを位置合わせの直後(立ち下がり同期)にすることで，パイプラインステージ数を5ステージから4ステージにすることも可能である．しかし，Jadeではクロックの立ち上がり同期にこだわり，結果として5ステージのパイプラインとなっている．

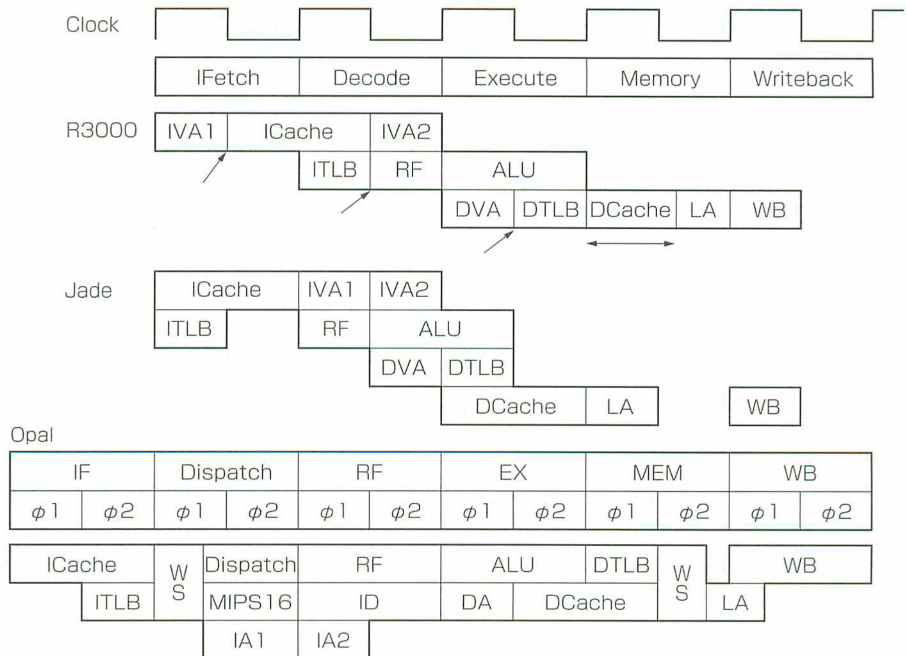


図14 R3000とJadeのパイプライン比較

● Jadeパイプラインの利点

Jadeパイプラインは三つの利点があるといわれている．一つ目は，キャッシュアクセスに余裕があること．二つ目は，クリティカルな操作がすべて立ち上がり同期になっているので，ある論理ブロックをユーザーが設計した論理に置き換えることが容易なことである．三つ目は，論理合成ツールによる遅延の調整が容易になることである．本来，論理合成を想定した機能設計は，クロック遅延のばらつき(クロックスキュー)を一定値内に収める操作を容易にするために，クロックの立ち上がりエッジのみを使用する．これを実践したわけだ．

MIPS社の発表によると，0.25μmプロセスで製造した場合の動作周波数は，最悪の場合(単純な論理合成)で100～150MHz，典型的な場合(専用設計)で150～255MHzだそうである．クリティカルな操作を立ち上がり同期にしたとはいえ，パイプライン効率はR3000のそれと大差がないのも事実で，この動作周波数が可能なのか否かは実際に回路設計した人にしかわからないだろう．

MIPS社は，Jadeの拡張版の4KEc(コードネームEmerald)でMIPS16命令セットに対応すると発表した．しかし，その実装方法たるや，1段のデコードステージ内でMIPS16からMIPS32へのコード変換を行ってデコードするという，非常に厳しいタイミングを提唱している．Jade開発当時の理念はどこへ行ってしまったのだろうか．もっとも，4Kcと4KEcの構造的な違いは，低消費電力を実現するゲーテッドクロックを行うか行わないかの違いだけである．

なお，先頃発表されたPSP(Play Station Portable)のCPUコアは4Kcまたは4KEcであって，R4000系ではない．

● Opalのパイプライン

Opalではさらにパイプラインが変更された．OpalのパイプラインはIF，Dispatch，RF，EX，MEM，WBの6ステージで構成され，φ1，φ2の2相クロックで動作するとされている．しかし，論理合成を容易にするために単相クロックを採用しながらも説明上の方便のためφ1，φ2を使用しているのではないかと思われる．

Opal自身はスカラプロセッサだが，スーパースカラへの移行の可能性を残している．つまり，ディスパッチステージが命令フェッチとレジスタリード/命令デコードステージの間に挿入された．このためパイプラインは，Jadeより1ステージ多い6ステージとなる．これは，将来的には，複数の演算ユニットに命令をディスパッチ(発行)するために使用する．命令デコード自体にも余裕ができるので，動作周波数が少し向上する．また，この追加ステージはMIPS16のためのプリデコードステージとしても利用できる．

パイプラインのステージ数が増加することで分岐の性能が悪くなるが，Opalでは静的な分岐予測と命令プリフェッチで対応している．分岐はすべてTAKENするものと仮定し，投機的に6命令をフェッチできる．分岐予測が外れた場合のペナルティは1サイクルにすぎないという（筆者としては懐疑的である）．Opalのパイプラインの詳細を以下に示す．

1）IF φ1＋φ2

命令キャッシュにアクセスする（ICache）．命令の仮想アドレスはDispatchステージ（IA1）とRFステージ（IA2）で計算される．

2）IF φ2

マイクロTLBにアクセスし，命令の仮想アドレスを物理アドレスに変換する（ITLB）．

3）Dispatch φ1＋φ2

命令キャッシュのヒット/ミスをチェックする（WS：Way Select）．スーパースカラ構成を採るための命令ディスパッチ用のタイミングを提供する（Dispatch）．MIPS16をサポートする場合のプリデコードタイミングを提供する（MIPS16）．次の命令のための命令の仮想アドレスを用意する（IA1）．

4）RF φ1＋φ2

レジスタをフェッチする（RF）．命令をデコードする（ID）．

5）RF φ1

分岐先の仮想アドレスを計算する（IA2）．

6）EX φ1＋φ2

演算を行う（ALU）．

7）EX φ1

ロード/ストア命令のオペランドアドレスを計算する（DA）．

8）EX φ2

データキャッシュへのアクセス（DCache）．1段目．

9）MEM φ1

オペランドの仮想アドレスを物理アドレスに変換する（DTLB）．データキャッシュへのアクセス（DCache）．2段目．

10）MEM φ2

データキャッシュのヒット/ミスをチェックする（WS）．データキャッシュからフェッチしたデータと，データキャッシュにストアするデータの位置合わせをする（LA）．

11）WB φ1＋φ2

EXステージでの演算結果をレジスタファイルに書き込む（WB）．ストア命令の場合はデータキャッシュに書き込む．

● JadeとOpalの性能……どちらが高い？

MIPSの発表によると，Opalを0.15 μmプロセスで製造した場合の動作周波数は450MHz，0.18 μmプロセスでは375MHzだそうである．Opalでは，Jadeでわざわざ立ち上がり同期に揃えたデータキャッシュアクセスが立ち下がり同期に変更されていることもあり，スーパーパイプライン構造も採用していないので，本当にこんな高周波数で動作可能なのかは不明である．

さらにMIPSの発表によると，JadeとOpalの性能（MIPS/MHz）は，どちらもDhrystone MIPSで1.2であるという．これはR3000とほぼ同じ性能である．Opalに関しては，パイプラインのステージ数が増えているのに，Jadeと同じ性能というのは納得がいかない．4KEcの発表ではMIPS/MHzを1.7であった．Jadeと実体は同じなのに，この性能アップの理由は不明である．そもそも1.7という値はスーパースカラでないと実現できない．

それはさておき，同じ性能のIPコアが二つも必要なのかという疑問は残る．MIPSの弁明では，Dhrystoneベンチマークでは真の性能はわからない，実際のアプリケーションではOpalはJadeの2倍の性能があるという．これはキャッシュ容量を2倍にできる点と，64ビット演算と32ビット演算の差と説明されているが……．

JadeにしろOpalにしろ，論理合成可能なRTL（Register Transfer Level）記述で提供されるのだが，目標動作周波数が達成できるか否かは，LSI製造メーカーの技術力によると思う．しかしMIPS社の説明では，JadeにしろOpalにしろ，何も特別なことを行っているわけではなく，誰が作っても200MHz以上の性能は保証するとしている．また，Dhrystoneベンチマークの値が異様に高い理由としては，アーキテクチャをDhrystoneに特化しているらしい．実アプリケーションではあまり効果がないが，Dhrystone MIPSの値は採用の決め手になることが多いので，あえてそのような構造にしているという．

5 SH-1/SH-2/SH-3，そしてSH-5

● 16ビット固定長命令RISC

SHシリーズはルネサステクノロジ（当時は日立製作



図15　SHのパイプライン
（小文字表記のステージはダミーステージ）

所）が1992年に発売した，組み込み用途をねらったRISC型32ビットマイクロコントローラとして誕生した．その後，積和演算やMMUを内蔵し，MPUとしての地位を確実にしている．多くのRISCが32ビットの固定長命令であるのに対して，SHシリーズは16ビット固定長命令を採用しコードサイズの削減を図っている．

SHシリーズはDLX（R3000）のパイプラインを参考にしているといわれるが，命令によってパイプラインのステージ数が異なっている点で，CISCの考え方を引きずっているようにも思える．

パイプラインは，次の5ステージから構成される．ステージ構成だけを見れば，R3000と同一である．また，遅延分岐は採用しているが遅延ロードは採用せず，データハザードが生じる場合はインタロックする．

1）IF：命令フェッチ
2）ID：命令デコード
3）EX：命令実行
4）MA：メモリアクセス
5）WB：ライトバック

IF，ID，EXの3ステージはすべての命令に存在するが，命令によっては，MA，WBステージがない場合もある．主なパイプラインを**図15**に示す．図を見るとわかるが，パイプラインはSH-1/SH-2とSH-3で少し異なっている．レジスタ-レジスタ間演算（転送を含む）は，SH-1/SH-2ではIF，ID，EXの3ステージで構成されるが，SH-3ではデータを保持するだけのMAステージと，レジスタへ値をライトするためのWBステージが追加されて5ステージ構成になっている．

私見だが，これらの命令において，SH-1/SH-2/SH-3とも，レジスタのリードはEXステージで行っているようである．そして，演算を行った結果は，SH-1/SH-2ではEXステージのうちに，SH-3ではWBステージでレジスタにライトするようである．

R3000のパイプラインを参考にした（といわれる）わりには，EXステージまでレジスタリードを遅延させたり，SH-1/SH-2では演算結果をEXステージでレジスタにライトさせたりするなど，タイミング的に厳しい設計になっている．これは，レジスタのフォワーディングをまったく行っていないか，フォワーディングの論理を軽くするためと推測される．もっとも，SH-4ではIDステージでレジスタをリードするようになったようで，試行錯誤の跡がうかがえる．

さて，ロード命令はパイプラインの5ステージすべてを使う．WBステージは，最初はロードしたデータをレジスタにライトするためだけに存在していたようだ．SH-3ではレジスタ-レジスタ演算にも適用された．一方，ストア命令はレジスタへのライトがないのでWBステージが存在しない．いずれにせよ，命令の種類に応じてパイプラインのステージ数を可変にするのはCISCの発想である．実質的にはパイプラインのスループットは，最大のステージ数に支配されるのであまり効果はない．

そのことに気づいてか，SH-3ではパイプラインがほとんどの命令で5ステージ固定に改善された．しかし，ストア命令がなぜ4ステージのままなのかは謎である．

● SH-5

SH-4では性能向上のためにスーパースカラ構成を採用したが，1999年に発表されたSH-5ではシングルパイプラインに戻した．400MHz動作を達成するためには，スーパースカラの制御の複雑さがスピード上のネックになるという理由からだ．

SH-5のパイプラインは，

1）Fetch-1（F1）：命令フェッチ
2）Fetch-Decode（FD）：命令フェッチ＆デコード
3）Decode（D）：デコード

(プリ)フェッチ	デコード	実行	メモリ	書き込み

(a) ヘネシー&パターソンのパイプライン

プリフェッチユニット	命令デコーダ	ALU/シフタ	メモリアクセス	
	レジスタリード	乗算器		レジスタライト

(b) ARM8のパイプライン

命令キャッシュ	命令デコーダ		ALU 乗算器	R	DC	R/S	
	レジスタリード	シフタ					レジスタライト

R:ローテート
DC:データキャッシュ
R/S:ローテート/符号拡張

(c) StrongARMのパイプライン

整数演算

BTB / Fetch1	Fetch2	Decode	Reg. File / Shift	ALU	State	WB

BTB　I-Cache　Decode　Register　Shift　ALU

ロード/ストア

BTB / Fetch1	Fetch2	Decode	Reg. File / Shift	ALU (Address)	DCache1	DCache2	WB

DCache

(d) XScaleのパイプライン

図16 ARM系プロセッサのパイプライン

4) Execute-1(E1):実行
5) Execute-2(E2):実行
6) Execute-3(E3):実行
7) Writeback(W):ライトバック

の7ステージで構成される.

E1, E2, E3ステージからDステージへのフォワーディングが可能だというが，だったらSH-4までは「フォワーディングを行っていなかったのか」と突っ込みたくなるが，RISCであるからには，そんなはずはないだろう.

パイプラインのステージ数の増加にともなう分岐命令の性能低下を補うため，SH-5ではSplit Branch(分割分岐とでも訳すか?)という方式を採用している. これは，分岐先アドレスを計算して命令プリフェッチを行っておき，実際の分岐命令でその命令を実行するという2段階の構造で分岐命令処理を実現する. そのためにPTA(Prepare Target Address)という命令が用意された.

1999年に発表されたSH-5であるが，2002年になって初めて展覧会などでプロトタイプのデモが行われるようになった. 動作周波数は400MHzにはほど遠い256MHzである. SH-6やSH-7についても計画が発表されているが，ここでは省略する.

6 ARM/StrongARM/XScale

● ARM7までとARM8

最初のARMアーキテクチャのMPUが開発された当時，RISCにはスタンフォード大学のMIPSと，カリフォルニア大学バークレー校のRISC I, II(SPARCの母体)しか例がなかった. ARMはバークレーRISCを参考にして設計された. ロード/ストアアーキテクチャ，32ビット固定長命令，3オペランドフォーマットという特徴を採り入れたが，レジスタウィンドウ，遅延分岐，全命令の1クロック実行は採用しなかった(ほとんどの命令は1クロックで実行するが). 設計目標は，CISCライクな命令セットをRISCに準じた単純なハードウェアで実行することに置いている. 命令セットの特徴はソースオペランドをシフトした後に演算可能なこと，ほとんどすべての命令が条件コードを変更し,条件コードに応じた処理が可能なことである.

ARMにはARM1～7，ARM8，StrongARMとアーキテクチャに若干の差異がある. ARM1～7は単純な3ステージのパイプラインを基本としていたが，その後は改良が重ねられ，ARM8で標準的な5ステージのパイプラインにたどり着く.

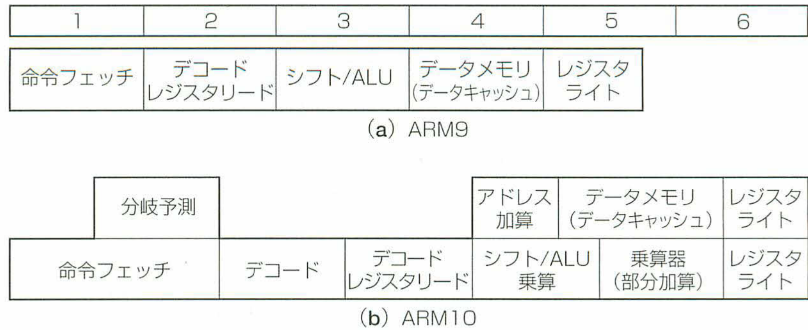


図17
ARM9/ARM10のパイプライン

ARM8では，パイプラインへの命令供給のバンド幅を向上させるため，命令のプリフェッチを行いバッファリングする．初代のARM8のプリフェッチユニットには静的な分岐予測機能も内蔵されていたという．**図16(b)**にARM8のブロック構成を示す．パイプラインは次の5ステージから構成される．

1) 命令プリフェッチ
2) 命令デコード，レジスタリード
3) 実行(シフトと演算)
4) メモリアクセス
5) ライトバック

● ARM9/ARM10

ARM8の後継であるARM9のパイプラインは，ARM8とほとんど同じである．その後継のARM10ではパイプラインに変更が加えられた(**図17**)．つまり，高い動作周波数を実現するために，デコード部と実行部のステージを2段に分割している．キャッシュアクセスは1.5段分をかけてアクセス時間に余裕をもたせている．また，アドレス計算用の加算器を専用に持ち，ロード/ストアのパイプラインを整数演算系と分離している．これにより，データキャッシュはノンブロッキング(ヒットアンダミス)が可能になっている．さらにARM10では，パイプラインのステージ数増加による性能低下(CPIの増加)を低減するため，動的な分岐予測が採用された．ARM社は，これによりARM10はARM9と同等なCPIが得られるとしている．

● StrongARM

ARMのパイプラインはARM社とDECが共同開発したStrongARM(現在はIntelに買収されている)で一応の完成をみる．キャッシュの構成が命令とデータに分割された(命令とオペランドフェッチで待ち合わせが生じない)こととレジスタのフォワーディング機能が追加されたのが特筆すべき特徴である．パイプラインは次の5ステージで構成される．

1) 命令フェッチ(命令キャッシュから)
2) 命令デコードとレジスタリード，分岐先のアドレス計算
3) オペランドのアドレス計算，またはシフトおよび演算を実行
4) データキャッシュへのアクセス
5) レジスタファイルへ結果をライトバック

図16(c)にStrongARMのパイプライン構成図を示す．

ARMのパイプラインも命令ごとに可変なステージ数から始まり，結果として5ステージに落ち着いたようである．やはり，5ステージというのがRISCのパイプラインの王道といえるのかもしれない(少なくともこれまでは)．

● XScale

Intelから発表されたXScale(かつてStrongARM2と呼ばれた)では，600MHz(当初の目標)という高い動作周波数を実現するため，再びパイプラインの見直しがされた．結果，整数演算で7ステージ，ロード/ストアで8ステージという構成になった〔**図16(d)**〕．ステージ数はそれほど多くはないが，インテルはこれをスーパーパイプラインと呼んでいる．

パイプラインが2ステージ増えた理由は，おもに2本のクリティカルパス(タイミングネックになる論理経路)対策のためである．一つ目はALU演算である．従来のStrongARMでは1クロックで

　シフト→ALU演算→条件コードの生成

を行っていた．これを3ステージに分割して処理する．こうすることにより，命令デコードにも余裕ができた．従来は命令デコードとレジスタアクセスを1クロックで行っていたが，

　レジスタアクセス→シフト

のタイミングを，従来より遅らせて，余裕をもたせている．

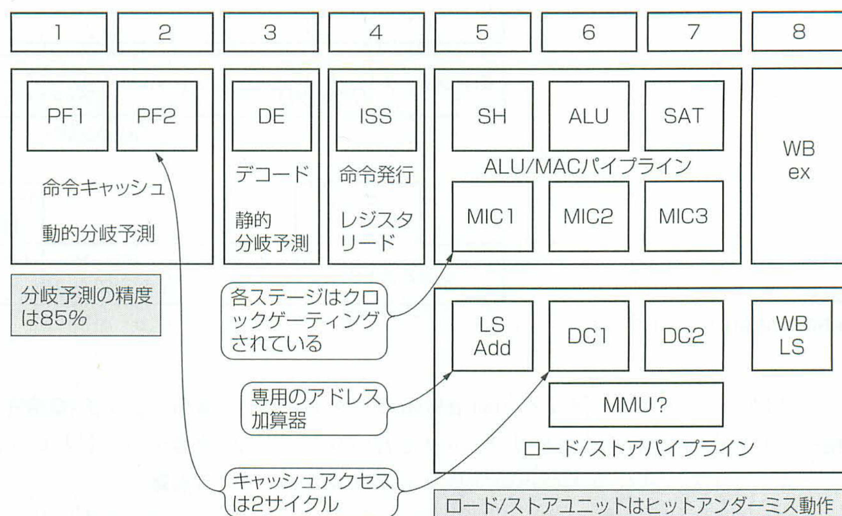


図18　ARM11のパイプライン

二つ目はデータキャッシュのアクセスである．従来は，データキャッシュを，

アドレスデコード→キャッシュアクセス→
データの整列→ALUへ入力

と1クロックで行っていた．XScaleではデータキャッシュが従来の2倍の32Kバイトになったので，一度に動作する回路が多くなりクリティカルパスになった．そこでデータキャッシュアクセスを2クロック(2ステージ)で行うように改良した．

XScaleではパイプラインのステージが増加したため，分岐命令の性能低下(当然，分岐予測機構は備えている)などを考慮するとCPIが5～8%増加するが，周波数を1.5倍に向上することで，差し引き40%程度の性能向上となる．なお，分岐ターゲットバッファは128エントリからなるダイレクトマップキャッシュで，2ビットの情報で分岐の履歴を管理する．

● ARM11

一方，ARM社は2002年4月にARM11の概要を発表した．8ステージのシングルパイプラインで350～500MHz動作を目指す．明らかにXScaleへの対抗策と見てとれる．**図18**にARM11のパイプラインを示す．ARMアーキテクチャのクリティカルパスは，XScaleでも説明したが，シフト＋ALUの同時実行，キャッシュアクセス，そしてMMUにある．これらのステージを独立化することで，高速動作を実現できる．基本的な考え方はXScaleのパイプラインとよく似ている．

なお，Thumb-2をサポートするARM11では，命令キャッシュとデコードステージの間に位置合わせのステージが追加されて9ステージのパイプラインとなっている．Thumb-2では32ビット長と16ビット長の命令の混在を許すので，命令の先頭アドレスを検出するためのステージである．

7　V800シリーズ

V800シリーズはNECがV80の後継として開発した，どちらかといえば，マイクロコントローラといえるMPUシリーズである．その開発目標は，低価格で低消費電力のチップであった．V800シリーズでは基本の命令長を16ビットとしながらも，大きいビット数のイミディエート値やディスプレースメントの指定でコード効率を上げるために，32ビットの命令長も用意している．ちょうどSHとARMの中間のようなアーキテクチャである．

V800シリーズは，1992年に最初のV810が開発され，その後V830，V850と続いて開発された．現在は，これらのMPUをCPUコアとした周辺内蔵品が販売されている．

V810のパイプラインに関しては，ユーザーズマニュアルには記載されていない．しかし，当時の雑誌の解説記事によると，

フェッチ，デコード，実行，書き込み

という典型的(と記述されている)なRISCのパイプラインに対し，デコードと実行の間にレジスタリードのステージを挿入した，

1) フェッチ
2) デコード
3) レジスタリード

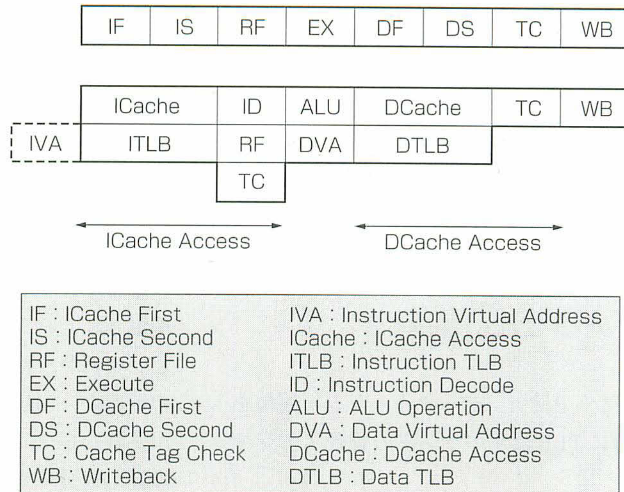


図19 R4000のパイプライン

4) 実行

5) 書き込み

という構成だという．これは，可変長の命令フォーマットをデコードするのに長い処理時間が必要であり，デコードに余裕をもたせるためと説明されている．合計5ステージ構成のパイプラインであるが，これは33 MHz以上の動作周波数を想定したものであり，25 MHzの動作周波数ではレジスタリードのステージは結果として不要だったようだ．

しかし，このパイプライン構成では，ロード/ストア命令でのオペランドフェッチステージがない．おそらく，ロード/ストア命令の処理時には6ステージになるのであろう．

その後，V830/V850になるとパイプラインの見直しが行われ，

1) IF：命令フェッチ

2) RF：命令デコード

3) EX：実行

4) MEM：オペランドアクセス

5) WB：ライトバック

という5ステージのパイプラインになった．これは何度も説明している典型的なRISCのパイプラインと同じで，取り立てていうこともない．ただ1点，分岐命令は，TAKENする場合，EXステージの終了を待ってIFステージを開始する．これは前半の**図9(c)**と同じである．明らかに分岐先のアドレス計算から命令フェッチまでに時間的余裕をもたせているのがわかる．このため，分岐命令のレイテンシは3クロック（NO TAKENの場合は1クロック）になる．

V800シリーズでは遅延分岐を採用しないため，分岐が多いプログラムの処理は不利になる．また，歴史的には古いMPUのせいか，分岐予測機構も採用していない．シンプルイズベストという考え方なのだろう．

8 R4000

R4000はスーパーパイプライン構造を採用し，高い動作周波数で動作させることを目的としている．パイプラインはIF，IS，RF，EX，DF，DS，TC，WBの8ステージで構成され，（筆者の予想では）単相クロックに同期して動作する．**図19**にR4000のパイプラインの詳細を示す．各ステージでの動作は次のようになっている．

1) IF

命令フェッチ1段目．命令の仮想アドレスが命令キャッシュとTLBに転送される．

2) IS

命令フェッチ2段目．命令キャッシュが命令を出力し，同時にTLBは命令の物理アドレスを出力する．

3) RF

レジスタファイル．次の3動作が並行に行われる．

a) 命令をデコードし，インタロック条件をチェックする

b) 命令キャッシュのヒット/ミスがチェックされる

c) レジスタファイルからオペランドをフェッチする

4) EX

命令実行．次の3動作の一つが実行される．

a) 命令がレジスタ - レジスタ間命令なら演算を実

行する

b) 命令がロード/ストア命令ならオペランドの仮想アドレスを計算する

c) 命令が分岐命令なら，分岐先の仮想アドレスを計算する．同時に分岐のTAKEN/NOT TAKENを決定する

5) DF

データキャッシュ1段目．オペランドの仮想アドレスがデータキャッシュとTLBに転送される．

6) DS

データキャッシュ2段目．データキャッシュが値を出力する．同時にTLBはオペランドの物理アドレスを出力する．

7) TC

タグチェック．ロード/ストア命令の場合，データキャッシュのヒット/ミスをチェックする．

8) WB

ライトバック．命令の実行結果をレジスタファイルに書き込む．ストア命令の場合はデータキャッシュに書き込む．

R4000の各パイプラインステージは基本的には1クロックであるが，時間がかかるキャッシュアクセスには時間をかけている(タグチェックを含めて3クロック)．R4000の発表当時はスーパーパイプラインとしてクローズアップされたが，現在においてはごく普通のパイプライン構成である．

R4000ではパイプラインが8ステージになったため，分岐命令の実行時に3クロック，ロード命令の実行時に2クロックの遅延スロットが生じる．分岐命令においてはR3000と互換性をもたせるため遅延スロットの1命令分は実行するが，残りの2クロックはバブル(むだな時間)になる．分岐命令の実行時間がR3000の1クロックから3クロックになった(遅延スロットを含まない)と思えばよい．

ロード命令においては遅延スロットに相当する後続2命令がロード命令のデスティネーションオペランドと一致している場合はインタロックが生じる．つまり，R4000では遅延ロードを採用しない．さすがに，ロード命令とその結果を使用する命令の間を2命令分も空けるのは現実的ではないと考えたのであろう．

分岐命令の実行時間を短縮するため，R4000ではLikely分岐(Branch Likely)が導入された．Likely分岐とは，分岐条件が成立するときのみ遅延スロットの命令を実行する条件分岐命令である．分岐条件が成立しなければ遅延スロットは無効化される．遅延スロットにNOP命令があると考えてもよい．分岐命令がループ処理の終わりにあるような場合，分岐命令をLikelyにして分岐先の1命令を遅延スロットに置けば，ループ内の命令が1命令分減少するので，実質的に分岐命令の実行時間を短縮できる．これは一種の(静的な)分岐予測とみなすこともできる．

Likely分岐はR4000以降のMIPSアーキテクチャで採用されているが，スーパースカラ構造では実装が難しいせいか，将来的には削除したい意向だという．その前兆か，MIPS-3Dという拡張アーキテクチャで採用された`bc1any2`，`bc1any4`という条件分岐命令ではLikely分岐が定義されていない(命令コードとしては割り当て可能)．

9 Topaz(24K)のパイプライン

それまでMIPS社はIPコアビジネスに注力してきたが，2003年6月にその方向転換を行った．従来，32ビットアーキテクチャのMIPS32と64ビットアーキテクチャのMIPS64の2系統のアーキテクチャを管理してきたが，64ビットアーキテクチャの需要がないのか，32ビットに特化するようになった．それまで，MIPS64系のIPコアとしては，5K(Opal)，20K(Ruby)，25K(Amethyst)が存在したが，最新のロードマップには5Kのみが掲載されている．従来，20Kや25Kがあった位置には，新たに24K(Topaz)が掲載されている．24Kという名称からも，性能的に25Kを置き換えるという意味合いが感じられる．

24Kは，6月17日に発表したMIPS32アーキテクチャの論理合成可能なIPコアである．8ステージのシングルパイプラインで動作周波数目標は400～500MHzである．最初からマルチプロセッサに対応し(キャッシュはMESIアルゴリズム)，マルチコアで性能向上を目指す．2004年からライセンスを開始する．

MIPS社から発表された24Kのパイプラインを**図20**に示す．パイプライン自体には不明な点が多いが，とりあえず発表された情報を説明する．

1) IF

命令キャッシュ第1段階．

2) IS

命令キャッシュ第2段階．最大64Kバイトの4ウェイキャッシュに対応(16Kバイト/1ウェイ)し，命令キャッシュは2サイクルのレイテンシ(RAMアクセス



図20 Topaz(24K)のパイプライン

に1サイクル，タグチェックとウェイ選択に1サイクル)でアクセスする．命令キャッシュからは2命令を同時にフェッチ可能で，結果は6エントリの命令キューに格納される．これにより，命令フェッチと実行がデカップル構成になり，高い命令発行レートを維持できる．最大2命令までのノンブロックリードが可能であり，命令キャッシュへのリフィル時間を最適化できる．動的な分岐予測(512エントリの分岐履歴テーブル)，4エントリのリターンスタックを備える．分岐予測ミス時のペナルティは4クロックである．

3) RF

レジスタファイルへのアクセス．

4) AG

アドレス生成および命令発行を行う．整数演算とメモリパイプラインを分離することで，データキャッシュは4エントリのノンブロックリードに対応する．なぜ整数パイプラインにアドレス生成が必要なのかは，不明(分岐アドレスの生成用か)である[注1]．

5) EX

命令実行．

6) MS

乗算命令とシフト命令処理用の追加ステージ．32ビット×32ビットの乗算が，1クロックのリピートレート，5クロックのレイテンシで実行できる．

7) ER

結果の整列(ローテート)と符号/ゼロ拡張を行うステージ．

8) DCache

データキャッシュへのアクセス．クリティカルワードをフォワーディング可能(キャッシュバイパス)．

9) Sel

データキャッシュのタグチェックとウェイ選択．

10) WB

結果をレジスタへライトバック．

24Kのパイプラインの特徴としては，将来の高速動作を見越して，命令フェッチとデコードをデカップル構成にしたことであろう．シングルパイプラインでは珍しい．

まとめ

シングルパイプラインの概要について説明してきた．思えばヘネパタは偉大だった．いうまでもないことだがHennessyはMIPS RISCの生みの親，Pattersonはバークレー RISCの生みの親である．この2人によって著わされたヘネパタ本は，日本のRISCメーカーの技術者に多大な影響を与えた．ルネサステクノロジのSHシリーズやNECのV800シリーズはヘネパタ本で示されたアーキテクチャ(とくにパイプライン)を参考にしているといわれている．それだけDLX(R3000)のパイプラインが洗練されているということなのであろう．パイプラインの実例も，はからずもMIPSアーキテクチャの例が多くなってしまった．なおヘネパタ本は，日本語訳も出ているので，未読の方は一度は目を通すことをおすすめする．

注1：MIPS社の説明によれば，スキュードALU(Skewed ALU：ロードデータを演算器に入力するタイミングを意図的に遅らせて，ロード時のインターロックを削減する手法)のために必要というが，具体的な動作は，やはり不明．

第3章

1クロックで複数の命令を同時に実行する

並列処理の基本とスーパースカラ

ここではシングルパイプラインを多重化したスーパースカラについて解説する．1命令1クロック処理が当たり前になってくると，プロセッサの性能はクロック数(と命令の機能)だけで決まってしまい，アーキテクチャ的には進化の余地はないように思える．そこで登場するのが1クロックで複数の命令を同時に実行してしまおうというアプローチだ．その代表がスーパースカラという手法であり，現在の高性能MPUの多くで採用されている．前半ではスーパースカラの基本的な考え方を解説し，後半では実際のプロセッサの実装方式を解説する．

スーパースカラの基本

1 CPIからIPCへ

CPI(Clock cycles Per Instruction)とは，1命令を実行するのに必要なクロック数である．この値が小さいほどMPUは高性能であるといえる．CISCからRISCへの進化によって，CPIが1という限界に達してしまった(理想的な実行環境に限定されるが)．それ以上性能を上げるには，同じパイプラインステージ内で複数の命令を実行させればよいと考えるのが自然な発想である．これがスーパースカラである．そうなってくると，性能指標としてCPIの逆数である**IPC**(Instructions Per Clock cycle)を使用したほうがわかりやすい．つまり，1クロックに実行できる命令数である．2命令を並列に実行できればIPCは2に近づくし，4命令を並列に実行できればIPCは4に近づく(理論的には)．

IPCは値が大きいほど高性能である．IPCは**MIPS**(Million Instructions Per Second)値とも密接な関係がある．MIPS値とは，1秒間に実行できる命令数(100万命令単位)である．その意味でいうと，IPCに動作周波数(MHz単位)を掛け算した値がMIPS値である．つまり，IPCが1の場合，100MHz動作では100MIPS，200MHz動作なら200MIPSである．

もっとも，最近のMPUのMIPS値は**Dhrystone MIPS**を採用しているので，公称性能は本来の意味のMIPS値とは異なる．Dhrystone MIPSとは，有名なミニコン(死語)であるVAX-11/780の性能を1MIPSとし，Dhrystoneベンチマークを実行したときの性能がその何倍の値になるかを示したものである．このDhrystone MIPSを用いれば，シングルパイプラインでもIPCが1を超えているように見えるので多用されている．

しかし，スーパースカラ構造になると事情が異なる場合もある．Dhrystone MIPSを用いると性能がそれほど高く見えないからだ．その代わり，IPCと同時実行できる命令の数が等しいと仮定して，たとえば，2命令同時実行可能なパイプラインを200MHzで動作させると400MIPS(200MHz×2命令という計算)などという理想値を示す場合もある．現実には，同時実行できる命令数を増やしていっても，IPCは1.6あたりに収束することが経験的にわかっているので，2命令同時実行でもIPCが2になることはまずない．

ただし，Dhrystone MIPSを真のMIPS値と(意図的に)混同してIPCを計算すれば，2命令同時実行で2.2程度になることもある．混同した場合でも，4命令同

時実行では4.0どころか3.0を超えることはまずない．その場合は，動作周波数×4でMIPS値が決められたりするのである．つまり，200MHzで動作し，4命令同時実行なら800MIPSといった具合である．まあ，公称MIPS値をそのまま信じる人はいないと思うが，このような数字のマジックに惑わされないようにしなければならない．

しかし，感覚的にはIPCが2.2などと言われると非常に高性能だと思ってしまう．現在のGHz単位で動作するx86系のMPUのIPCは2〜3などといわれているが，Dhrystone MIPSによるIPCでは0.6程度である(つまり実質的な性能は，動作周波数の割には高くない)．

一般にパイプラインのステージ数を増やすと，IPCは低下する．動作周波数を向上させるためにパイプラインのステージ数を増やすことはよくある手法だが，パイプラインのステージ数を増やしてもIPCを0.6程度に保ち続けているIntelやAMDは賞賛に値する．NetNews(fj.comp.arch)に，Pentium Ⅲ-750MHzでDhrystoneベンチマークを行った場合の性能の実測値が報告されていた(メッセージIDは失念した)．その値から計算すると，真のIPCは1.01，Dhrystone MIPSによるIPCが1.18であった．予想の2倍の性能になっているが，これはDhrystoneという，最高性能を発揮しやすいプログラムの性質によるものだろう．実際のアプリケーションではこうはいくまい．

ちなみに，別の資料によるPentium(P5)66MHzのDhrystoneによるIPCは1.5なので，Pentium Ⅲになると IPCは低下している．パイプラインのステージ数が増加しているので，当然といえば当然か．

とにかく，シングルパイプラインの目標がCPIを1に近づけることであったように，スーパースカラの目標はIPCを同時実行できる命令数に近づけることである．まあ，x86は独自の道を歩んでいるようにも思えるが．

2 複数の命令を並列実行する スーパースカラの概念

複数の命令を並列実行する機構を**スーパースカラ**(superscalar)と呼ぶ．スーパースカラでは並列に実行できる命令数をウェイと呼ぶ．イシュー(issue：発行)と呼ぶ場合もある．厳密には命令デコーダから複数存在する命令実行パイプラインに同時に送り込む(発行)ことのできる命令数がイシューであり，命令実行パイプラインの本数がウェイである．しかし，現在ではそれほど厳密には区別されていない．どちらかといえば，ウェイという表現のほうがよく使われている．

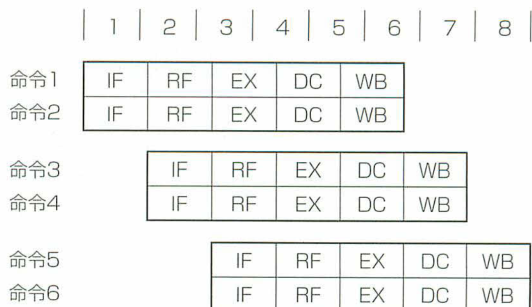


図1 スーパースカラ(2ウェイ)の概念図

一般に，2ウェイスーパースカラといえば2命令を並列実行できるパイプライン構造のことである．しかし，アウトオブオーダ実行が当然のようになっている現在の技術では，複数存在する演算器に対して2命令を同時発行できるパイプライン構造(2イシュー)のイメージのほうが強い．

いずれにしろ，スーパースカラの概念は**図1**のようなパイプラインの図で表されることが多い．つまり，命令フェッチ，命令デコード，実行，メモリアクセス，ライトバックを2命令並行に処理する，というイメージである．実際の動作とはあまり一致していないが，直感的ではある．

連続する命令は互いに独立しているのではなく，相互に関係している場合がある．このため，単純に命令の並列実行はできない．因果律が逆転するからだ．スーパースカラの最大の特徴は，MPUが複数の命令を並列に実行するからといって，プログラムで特別な考慮をする必要がないことである．従来からの命令セットを変更する必要もない．MPU自身が命令間の依存性を検出し，並列に実行可能な命令を自動的に判定し，演算器に対して発行する．そして各演算器は命令を並列に実行する．

もっとも，命令間に依存関係があると，処理にオーバヘッドが生じ，実行効率が低下するので，スーパースカラの真の性能を発揮するには，プログラム側での考慮(コンパイラによる命令の並び替え)が必要である．このため，新しいMPUが発表になると，従来のオブジェクトコードそのものではそこそこしか速くならないが，新しいMPU用に開発されたコンパイラで再コンパイルすると性能が劇的に向上する，ということがよくいわれる．

3 スーパースカラの実現

一般に，命令はデコーダでの発行，演算器での実行，実行の完了の過程を経て処理される．命令の発行はプログラムに書かれた順序で行うこともできるし，矛盾を生じない限りは，プログラムの順序を無視して行うこともできる．また，命令の実行は基本的に1サイクルなので，通常は発行された順序で完了する．ただし，実行クロック数の異なる命令を同時に発行すると，完了する順序が入れ替わることもある．当然，プログラムの順序で完了するとは限らない．処理がプログラムの順序どおりであることをインオーダ，プログラムの順序と異なることをアウトオブオーダと呼ぶ．

スーパースカラの方式は，プログラムの発行，完了が，それぞれ，インオーダかアウトオブオーダであるかによって4種類に分類できる．以下は簡単のために，2ウェイのスーパースカラを想定して説明する．

● 命令デコード

命令デコードは，インオーダ/アウトオブオーダでそれほど大きな違いはない．命令キャッシュから2命令(たいていの場合はウェイの数と同じ数)をデコードし，命令キューに入れて終了である．デコーダと演算器の中間に命令キューをもつ方式では，デコードと発行を独立に行えるので，(命令キューに空きがある限り)1サイクルごとに2命令をデコードできるので効率がよい．また，逆に考えると，命令キャッシュの参照が少々もたついても，その時間的なロスを命令キューで吸収して見えなくすることが可能である．事実，MIPSのRuby(20Kc)はWay予測を行って命令キャッシュを参照しているが，予測失敗時のペナルティは命令キューで吸収できると説明している．

命令キューの役割は，デコードと命令実行開始までの待ち時間を最小にすることもあるが，とくにスーパースカラにおいては命令間のオペランドの依存関係を調べることである．たとえば，片方の(先行する)命令の実行結果を，もう片方の(後続する)命令がソースオペランドとして使用する場合に依存関係があるという．簡単にいうと，レジスタ間のハザードである．もし，2命令間に依存関係がなければ，同時に実行可能なので，演算器に2命令(ウェイの数)を発行する．命令の追い越しを許さない場合(つまりインオーダ)は，デコードしている2命令間でのみ依存性を調べればよいので，いちいち命令をキューに入れなくてもデコーダのみの検査でこと足りる．このため，インオーダのスーパースカラ構造では，命令キューをもたないものも多い．ただし，依存関係がある場合はデコーダで(依存関係が解消するのを)待ち合わせることになるので，各サイクルで，常に2命令をデコードすることはできない．このため，少し効率が悪い．

なお，このような役割をする命令キューは特別にリザベーションステーション(Reservation Station)，または集中命令ウィンドウ(Central Instruction Window)などと呼ばれる．

また，命令の依存関係は命令キューで解消されているので，いったん演算器で実行が開始されるとレジスタ間のハザードによるストールは発生しない．命令ごとに定められた実行クロック数(レイテンシ)を経て実行が完了する．ただし，データキャッシュアクセスによるストールは発生する可能性がある．多くの場合，

Column 1　スーパースカラという名前の由来

ここでスーパースカラの名前の由来に触れておこう．スカラから連想されるのは，ベクトル量に対するスカラ量である．つまり，科学計算でおなじみのベクトルや行列演算に特化した並列処理ではなく，スカラ量に対する並列処理という意味でスーパースカラと呼ぶという説が有力である．この意味で，通常のシングルパイプラインをスカラパイプラインと呼ぶこともある．

また，スーパースケーラという呼び方もある．これは，1クロックで1命令を実行するという直感的な基準(スケーラ)を超えるという意味からきているらしい．この説はあまり聞いたことはないが，技術解説で有名な某誌ではそう説明されている．

とどのつまり，スーパースカラの語源ははっきりしない．ただ，最近の論文ではスーパースカラの反意語としてユニスカラ(uniscalar)が使用されるが，これなどは「スカラ＝パイプラインの本数」という概念からであろう．

スカラかスケーラかというのは，個人的には単なる発音の問題だと思う．英語による発音はスーパースケーラに近い(少なくともあのHennessy教授はそう発音していた)のだが，最近ではスーパースカラと表記されるほうが多いように思う．実際にスーパースカラと発音する外国人も多くなった．

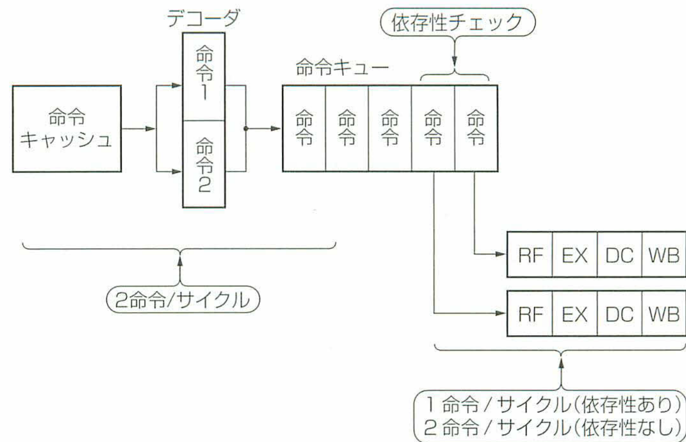


図2　インオーダ発行(2ウェイ)

データキャッシュにアクセスするためのロード/ストアユニットは実行ユニットと分離されているので，アウトオブオーダの場合は他の命令実行に影響を与えない．インオーダの場合は後続命令が先行するロード/ストア命令を追い越せないので，データキャッシュにアクセス中はパイプラインがストールしてしまう．

ところで，本来RISCは複雑な処理を行う命令を扱わないはずだったが，他社との差別化を進めるうちに複雑な命令も扱うようになってきた．これは，RISCのCISC化に通じる．複雑な命令に関しては，x86プロセッサではマイクロコードで処理するが，たいていのRISCはハードワイヤードロジック(結線論理)で処理する．しかし，これはハードウェアの複雑化を招く．そこで考案されたのが，複雑な命令は複数の単純な命令の組に分割して実行する方法である．これは，x86プロセッサがx86命令をμOPに変換して実行するのと同じ考え方である．たとえば，2001年に発表されたIBMのPower4は，複雑な命令を2命令(Cracking)または3命令以上(Millicode)に分割して命令キューに格納する．そして，分割された命令間で使用できる一時レジスタを4本，プロセッサアーキテクチャのレジスタとは別に備えている．今後このような傾向は増えていくかもしれない．

● インオーダ発行

この場合は，命令キューの先頭の(または現在デコードしている)2命令(ウェイの数)のみの依存性を調べる．命令間に依存性がない限り，2命令(ウェイの数)を同時に発行する．依存性がある場合は1命令のみを発行する．残った命令はその次の命令と組になり，再び依存関係が調べられる(**図2**)．

● アウトオブオーダ発行

この場合は命令キュー全体(あるいは一定の命令数の間)で依存性が調べられる．オペランドの依存性がない(というか，オペランドをすぐに利用できる)命令のうち，先頭から2命令(ウェイの数)を同時に発行する．発行される順序はプログラムの順序と入れ替わる場合がある．命令キュー内のすべての命令に何らかの依存関係がある場合は，先頭の1命令のみが発行される(**図3**)．

オペランドを利用可能かどうかは，リザベーションステーション内の各命令のソースオペランドとデスティネーションオペランドのレジスタ番号を比較することで検出できる．オペランド間に依存性がある場合でも，デスティネーションオペランドが確定していれば(そのデスティネーションオペランドを有する命令の実行が終了していれば)，ソースオペランドは利用可能と判断される(**図4**)．

● アウトオブオーダ完了

基本的にはRISCは命令を1クロックで処理できるが，現実には実行に数クロックかかる命令も存在する．とくに浮動小数点命令は実行に最低でも3クロック程度かかるのが実情である．つまり，MPU内には複数の演算器が存在するが，それらが処理する命令のレイテンシは一般には異なる．ということは，命令がインオーダに発行されようがアウトオブオーダに発行されようが，実行がプログラムの順序で完了する保証はどこにもない．すなわち，スーパースカラではアウトオブオーダ完了が自然な姿なのである(**図5**)．また，実行が終わった演算器には命令キューから次々と命令を発行すればいいので効率的である．

しかし，実行の完了と同時に結果をレジスタファイ

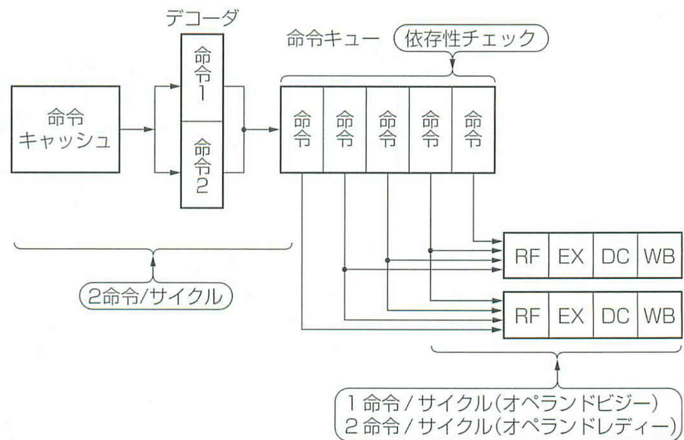


図3　アウトオブオーダ発行（2ウェイ）

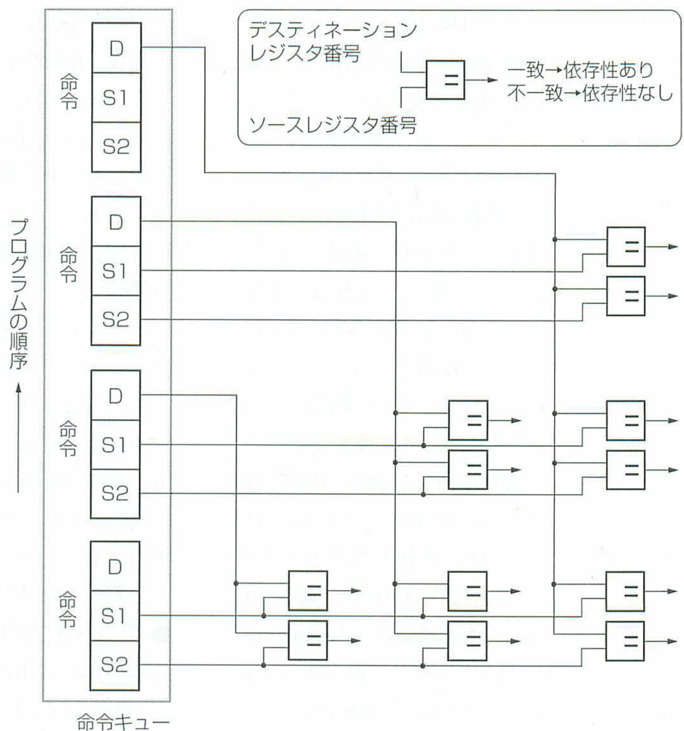


図4　オペランドの依存性チェック

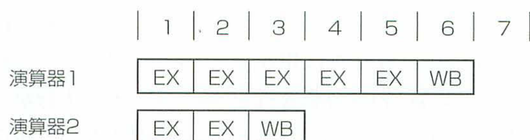


図5　アウトオブオーダ完了（2ウェイ）

ルに書き戻していたら不都合が生じる場合がある．

まず，第1は出力依存関係である．同時に実行されている命令のデスティネーションレジスタ（結果の格納先）が等しい場合，そこにはプログラム的に後にある命令の実行結果が書き込まれなければならない．ところが，後続命令の処理が先に完了し，先行命令の処理が後から完了する場合，正しい結果（後続命令の結果）が破壊されてしまう．これがWAW（Write After Write）ハザードである．シングルパイプラインではWAWハザードは起こり得ないが，スーパースカラでは当たり前に発生する．もっとも，これはあとで説明するレジスタリネーミングで回避することができる．

しかし，インオーダ発行のスーパースカラでは（回路規模が増大するのを嫌って）レジスタリネーミングを行わないことも多く，この場合はデコード時に発行を待ち合わせたり，後続命令のライトをストールさせるなど，何らかの対策が必要である．

第2は（こちらのほうがもっと深刻だが），例外の正確性（precise）の問題である．例外はプログラムの順序で処理されなければならない．たとえば，先行する命令も後続する命令も例外を発生する場合，後続命令の例外が先に検出されても，先行命令の例外発生を優先させるようにしなければプログラム処理に矛盾が生じる．

しかし，例外をプログラムの順序で発生させるという制約はシングルパイプラインでも同様であり，通常何らかの対策が施されている．問題は例外（割り込みでも同様）発生後に，例外の発生直後の命令からプログラムの処理を再開させる場合（ブレークポイント命令，システムコール命令，トラップ命令，割り込みなどの処理）に生じる．つまり，レジスタへのライトがプログラムの順番を無視して行われていたら，例外からの再開をどの命令から開始してよいのかが判断できない．

たとえば，op1，op2を適当な演算として，

R1 ← R2 op1 R3 ……命令1
R3 ← R4 op2 R5 ……命令2
例外/割り込み ……命令3

という命令処理を考え，op2の実行がop1よりも早く終了すると仮定する．この2命令の処理中に後続命令で検知される例外や割り込みが発生すると，R1は更新されていないのにR3が更新されている状況が発生する．この場合，プログラムの実行再開は，まだ実行されていない命令1から行うことになる．しかし，命令1のソースオペランドであるR3は命令2で更新される前の値が必要なので矛盾が生じる．例外の正確性を維持するのは，シングルパイプラインではそれほど複雑な制御でないが，（アウトオブオーダ完了の）スーパースカラではかなり複雑である．

例外が発生すると，それを致命的とみなし，プログラムの実行を中断する（再開しない），割り込みの受け付けは再開に都合のいい時点の処理が終了するまで待ち合わせる，という制御を行えば，アウトオブオーダ完了を実現できる．ただし，システムコールが行えないとか，割り込み応答性が悪くなるという問題が生じ，あまり現実的ではない．

● インオーダ完了

インオーダ完了とは，各演算器の完了がアウトオブオーダに完了するのは避けられないので，その結果を，いったん別の場所に保存しておき，レジスタにライトする順番をプログラムの順番に一致させる方式である．レジスタへのライトが終了するときに初めて命令は真の完了となる．この場合，演算器での実行完了と，命令の真の完了を区別する必要がある．一般には，前者をコンプリート（complete），後者をリタイアメント（retirement）と呼ぶ．リタイアメントはコミット（commit）と呼ばれることもある．なお，本書ではリタイアメントという表現は文字数が多いので，その動詞形のリタイアという表現を使用する．

インオーダ完了を実現するために，リオーダバッファ（Reorder Buffer：並び替えバッファ．ROBと省略）という機構が導入されている．リオーダバッファとは，プログラムの実行順序を記憶しておくテーブルであり，命令の発行時に適当な情報が設定される．その各エントリは，命令がコンプリートしたか否かの情報，命令の実行結果を一時退避するバッファ（このバッファはROBにない場合もある）などからなる．ROBはリザベーションステーションで共用することも可能である．

ROB内にある命令の先頭から，連続してコンプリートされている命令がリタイアできる（**図6**）．1サイクルにリタイアできる最大命令数はMPUごとに異なるが，多くの場合，スーパースカラのウェイ数に等しい．たとえば，命令1，命令2，命令3，命令4がプログラムの順序であり，これらの命令はすべてアウトオブオーダ（である必要もないが）に発行されているものとする．このとき，ROBの内容（先頭4エントリ）が，

命令1　未コンプリート

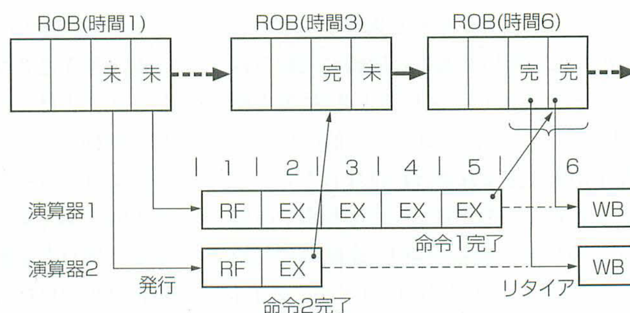


図6　インオーダ完了(2ウェイ)

　命令2　コンプリート
　命令3　コンプリート
　命令4　…

となっている場合，このサイクルでは1命令もリタイアできない．命令1が命令2以降のリタイアを阻害するからである．一方，

　命令1　コンプリート
　命令2　未コンプリート
　命令3　コンプリート
　命令4　…

となっている場合は，このサイクルでは命令1のみがリタイアできる．命令3のリタイアは命令2がコンプリートしていないので阻害されている．また，

　命令1　コンプリート
　命令2　コンプリート
　命令3　コンプリート
　命令4　未コンプリート

となっている場合は，命令1，命令2，命令3がリタイア対象である．ただし，実際にリタイアできる命令の最大数はMPUごとに異なる．もし，MPUが1サイクルで2命令がリタイア可能なら，命令1，命令2のみがリタイアし，命令3は次のサイクルでのリタイアに回される．もし1度に4命令がリタイア可能なら，命令1，命令2，命令3のすべてがリタイアできる．

例外の正確性の問題があるので，特殊な場合を除き，アウトオブオーダ完了というしくみは採用されない．したがって，スーパースカラの種類は，実質的には，インオーダ(インオーダ発行，インオーダ完了)とアウトオブオーダ(アウトオブオーダ発行，インオーダ完了)の2種類しかない．**図7**に典型的なスーパースカラ構成のMPUのブロック図を示す．**図7(a)**ではリザベーションステーションは一つのみであるが，**図7(b)**のように(いくつかの)演算器ごとにリザベーションステーションを設ける構成もある．

また，インオーダなスーパースカラは2ウェイのものが主流である．インオーダで3～4ウェイというのは筆者の記憶にない．これは，多ウェイのスーパースカラ構造を採用する場合でも，整数ALUは2個程度しか用意されていないためではないだろうか．インオーダのスーパースカラではウェイの数だけALUがないとパイプライン効率が悪い．それなら，いっそアウトオブオーダにしたほうが同時発行の効率が上がる．

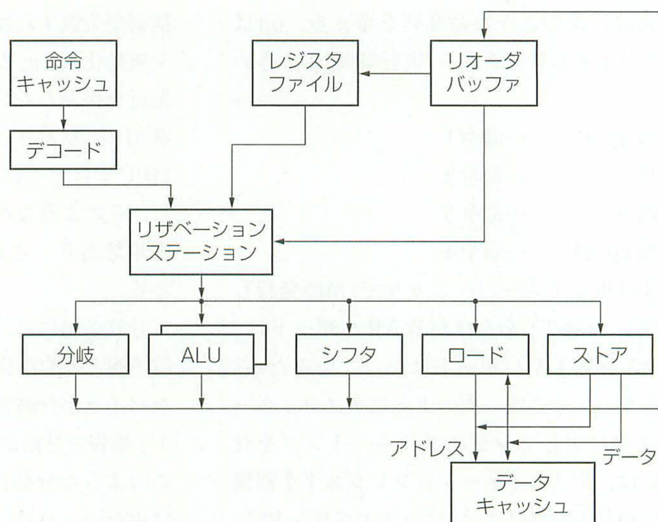
● 制限付きアウトオブオーダ

複数の演算器それぞれにリザベーションステーション(命令キュー)を有するスーパースカラ〔**図7(b)**のような構成〕において，各命令キューへの命令格納は独立に行われるのが普通である．しかし，すべての命令キューに空きができるまで命令の格納を待ち合わせてから，一括して命令供給を行う方式もある．命令のリタイアもすべての演算器での実行が終了してから一括して行う．この方式は，演算器の数が少ない場合はインオーダ方式と大差はないので，効率的でないが，命令キューの待ち合わせ論理を簡略化できるという利点がある．

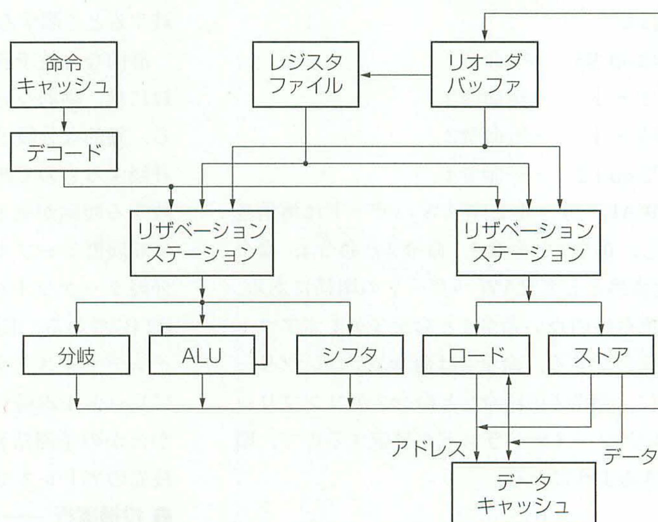
IBMのPower4では，ハードウェアを簡略化して動作周波数を向上させるため，4命令(＋1分岐命令)を一括してそのグループ単位で実行する方式を採用している．1グループ内の4命令はアウトオブオーダに実行できる．

4 スーパースカラの命令発行を効率的に行うための「レジスタリネーミング」

レジスタリネーミング(Register Renaming)とは，その名称のとおりレジスタ名の付け替えである．その役割には二つある．基本的には，スーパースカラの命令発行を効率的に行うための技術である．



(a) 単一のリザベーションステーションを有する場合



(b) 複数のリザベーションステーションを有する場合

図7　典型的なスーパースカラ構成

第1は，アーキテクチャ的に定義されたレジスタ数を増やすことである．たとえば，x86の系MPUの汎用レジスタは8本しかないので，ちょっとしたプログラムでもレジスタの使い回しが多くなり，レジスタの依存関係が発生しやすい．これは命令発行の制約となる．レジスタの本数をアーキテクチャが規定するより大きくもち，レジスタの名前を付け替えることで，内部的にプログラムの依存関係を低減できる．

第2には，これも同じく依存関係の解消であるが，WAR(Write After Read)ハザード，WAW(Write After Write)ハザードという偽の依存関係を解消することである．偽の依存関係とは，本来はハザードになるが，レジスタリネーミングによって依存性を解消でき，結果として命令発行の妨げとならないように変更可能な依存関係である．なお，WARとは先行する命令のソースオペランドを後続の命令で変更する可能性のある依存関係，WAWとは後続命令が変更したデスティネーションレジスタを先行する命令が変更する可能性のある依存関係である．これらは同一のレジスタが同時に変更される場合に生じるので，その同一のレジスタを別々のレジスタに割り当ててやれば(偽の)依存関係がなくなる．

また，真の依存関係とはRAW(Read After Write)ハザードのことであり，後続命令が先行する命令の実行結果をソースオペランドとして利用する場合である．これはレジスタリネーミングによっても解消でき

ない．たとえば，次のような命令列を考える．opは単純な加算(+)よりもレイテンシ(実行時間)の大きい演算とする．

R3 ← R3 op R5　……命令1
R4 ← R3 + 1　……命令2
R3 ← R5 + 1　……命令3
R7 ← R3 op R4　……命令4

この4命令は(4ウェイスーパースカラで)同時発行しようとしても，命令2と命令3がWARハザードに，命令1と命令3がWAWハザードになっているため，同時発行できない．そこで，次のようにデスティネーションレジスタに対してレジスタリネーミングを行う．基本的には，デスティネーションレジスタを別個のレジスタに割り当てればよい．ソースオペランドは，デスティネーションレジスタの割り当てにしたがって適宜変更される．

P1 ← R3 op R5　……命令1
P2 ← P1 + 1　……命令2
P3 ← R5 + 1　……命令3
P4 ← P3 op P2　……命令4

このとき，WARハザードとWAWハザードは解消される．しかし，命令1と命令2，命令3と命令4，命令2と命令4は依然としてRAWハザードの関係にある．この場合，依存性のない命令1と命令3がまずアウトオブオーダ発行できる．命令2は命令1がコンプリートするときに，命令4は命令2と命令3がコンプリートするときに，ソースオペランドが確定するので，晴れて発行できるようになる．

5 分岐予測と投機実行

● 分岐予測

典型的なプログラムでは全体のコードの10％が無条件分岐命令，10～20％が条件分岐命令であるといわれている．無条件分岐は，フェッチするアドレスを分岐先に切り替えるだけなので，それほど問題はない．一方，条件分岐は命令がパイプラインの実行ステージでコンプリートするまで，分岐するか否かが不明なのでやっかいである．分岐命令のコンプリートを待っていたのでは，その間に多くの命令をフェッチし，発行する機会を失うことになる．

そこで考案されたのが，分岐するか否かを推測するアルゴリズムである．もし，推測が成功すれば，命令はほんの少しの遅延(あるいは遅延なし)で続行できる．推測が失敗すれば部分的にコンプリートしている命令を無効化し，正しいアドレスからフェッチ，デコード，発行を再開しなければならない．これは，最近のx86系MPUのようにパイプラインのステージ数が多いMPUではとくに，かなりの性能低下をまねく．しかし，そのようなペナルティを考慮しても，**分岐予測**は必須であり，これを行わないと性能は悲惨なことになる．

分岐予測には二つの基本的な手法がある．静的な分岐予測と動的な分岐予測である．静的な分岐予測は，コンパイラが分岐命令の命令コードに埋め込んだ「ヒント」情報で分岐が発生するか否かを予測する．ただし，このような分岐命令を命令セットとして有するMPUは少ない．あるいは後方(backward)への分岐(オフセットが負)はループの終端とみなせるので，これを分岐すると予測するのも静的な分岐予測といえよう．

静的な分岐予測と動的な分岐予測を比較すると，一般には，動的な分岐予測のほうが効果的といわれている．動的な分岐予測とは，分岐命令の時間的な挙動を評価するものである．一度分岐した分岐命令は次も分岐する傾向があると予測する．これに使われるのは，分岐履歴テーブル(Branch History Table：BHT)と分岐ターゲットバッファ(Branch Target Buffer：BTB)である．BHTもBTBも分岐命令のアドレスをインデックスとするキャッシュである．(キャッシュにヒットする場合)BHTの出力は分岐命令が分岐するか否かの予測情報であり，BTBの出力は予測した分岐先のアドレスである．

● 投機実行 ―― 分岐予測の効果を増大させる

また，分岐予測の効果を増大させるために**投機実行**(Speculative Execution)を行うMPUもある．投機実行とは，分岐予測の成功/失敗がわかる以前でも命令を実行してしまう機能である．しかし，MPUは投機的に実行されている分岐命令の分岐/不分岐が確定するまで(つまり分岐命令のコンプリートまで)リタイアできない．もし分岐予測が失敗すれば，分岐命令以降に実行された命令を放棄して，分岐元から命令の処理をやり直さなければならないからである．通常，投機実行中の命令の結果は，リオーダバッファに格納される．分岐予測が失敗したらリオーダバッファの該当エントリを無効化すればよい．

ところで，投機実行に限らず，一般的には演算結果はリオーダバッファに格納されてリタイアを待つが，MIPSのR10000などは(レジスタにネーム後の)物理

レジスタを直接更新する．これは，アーキテクチャ上の論理レジスタにも実体があり，物理レジスタは値を一時的な演算結果を保持するものとして区別しているためである．この場合，リオーダバッファの中には演算結果の格納領域は不要である．R10000では命令のリタイア時に，その命令に割り付けられている物理レジスタの値が論理レジスタに転送される．

歴史的にはR10000方式のほうが古く，かつ一般的のように思える．x86系のMPUのように物理レジスタを実質的なレジスタの本数を増加させる目的でレジスタリネームを使用すると，それを一時的な結果の保存場所に利用することはできない．リオーダバッファ内に一時的な結果をもつのは姑息な方法のようにも思える．x86系MPUのシェアは膨大なので，そちらの方式が“大勢”といわれれば確かにそうではあるが．

投機実行を行う場合，分岐条件未確定中のロード/ストアがどのように処理されるかは興味深い．たとえば，ストアを行った後で分岐予測の失敗が判明したとき，キャッシュやメモリに不正なデータが書かれることはないのか不安になる．ロード/ストアがキャッシュ領域に対して行われるものならば，ROBと同等な一時バッファを設けて，ロード/ストア命令のリタイアまで保持すればよい．PentiumではこのバッファをMOB（Memory Order Buffer）と呼んでいる．

ロード/ストアが非キャッシュ領域に対して行われるときは事情が異なる．最近のMPUは専用のI/O命令をもっていないため，メモリ空間にI/Oアドレスを割り付けて，そこを非キャッシュで参照してI/O機能を実現する（メモリマップトI/O）．I/O装置にはリードを行うと内部状態が変化するものもあり，実際には実行されない（分岐予測が失敗する場合の）投機実行中のロードを行うと周辺が誤動作してしまう．つまり，このような場合，投機実行中の非キャッシュ領域へのロード/ストアは実行してはならない．また，同様に，非キャッシュ領域へのロード/ストアの順序も変更してはいけない．

最近のMPUは，ノンブロッキングキャッシュ機能を実装し，ロード/ストアもアウトオブオーダに行われるが，これはキャッシュ領域に対する場合のみである．

スーパースカラの実際

図8に，レジスタマップ，アウトオブオーダ命令発行，リタイアとリザベーションステーションの関係図（概念図）を示す．これらが，典型的なMPUにおいてどのように実現されているか，その実装方式を見ていこう．

各MPUとも，パイプラインに関しては実行ステージ以降の処理はやや枯れた感じがあり，新規性はない．しかし，命令フェッチ，分岐予測，命令デコードの部分には，各社それぞれにいろいろな工夫が見られる．そこを中心に比較するのも興味深い．

6 SH-4

● インオーダ/2ウェイスーパースカラ構造

SH-4は，整数ユニット，浮動小数点ユニット，ロード/ストアユニット，分岐ユニットという，四つの基本演算ユニットを備える．この四つのユニットに対し，2命令を同時発行するインオーダな2ウェイスーパースカラ構造である．なお，ロード/ストアユニットは単純な整数ユニットの役割をもち，簡単なMOVやNOPなどの命令を0レイテンシで実行できる．

このほかに，複雑な命令を実行するためのユニット（というか上述のユニットを組み合わせて利用？）があるようである．ユーザーズマニュアルによると，SH-4の命令は利用する内部機能ブロックにより，次の6グループに分類できる（略語の意味は筆者の推測によるもの）．

1) MT（Manipulate T）グループ
2) EX（Integer Execution）グループ
3) BR（Branch）グループ
4) LS（Load/Store）グループ
5) FE（Floating Point Execution）グループ
6) CO（Complex）グループ

これらのグループは上述の演算ユニットに対応しており，同時に実行できる組み合わせは**表1**のようになっている．CPUコア内の詳細な内部ブロック図は公開されていない．**図9**は筆者の想像図である．なお，各パイプラインは通常，次の5ステージで処理される．

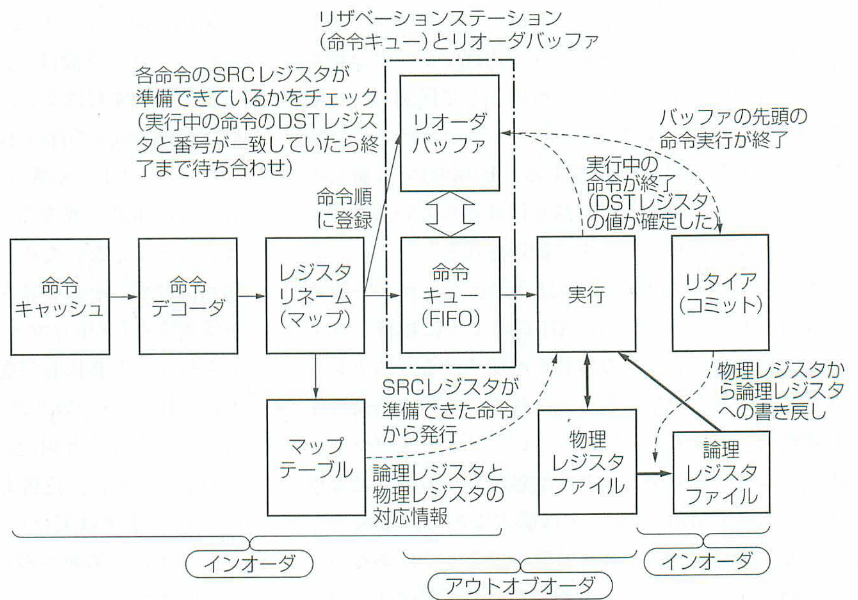


図8
スーパースカラ実現の概念図

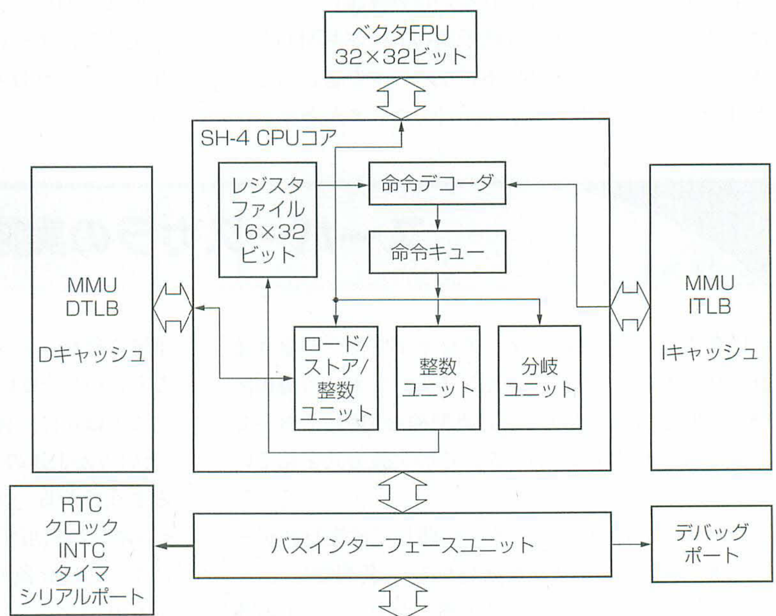


図9　SH-4のCPUコアの構造
(筆者想像による)

表1　SH-4の命令の並列実行性

		第2命令					
		MT	EX	BR	LS	FE	CO
第1命令	MT	○	○	○	○	○	×
	EX	○	×	○	○	○	×
	BR	○	○	×	○	○	×
	LS	○	○	○	×	○	×
	FE	○	○	○	○	×	×
	CO	×	×	×	×	×	×

1) 命令フェッチ(I)
2) デコード・レジスタリード(D)
3) 実行(EX, SX, F0, F1, F2, F3)
4) データアクセス(MA, NA)
5) ライトバック(S, FS)

これ自体はSH-3のパイプラインと同じである(浮動小数点演算が追加されているが).**図10**に基本的なパイプラインの命令処理パターンを示す.

ユーザーズマニュアルによると，2命令を同時発行できる場合は，0レイテンシと1レイテンシの命令の

組み合わせだけのようである．片方が0レイテンシならば無条件に，1レイテンシどうしなら，命令のグループが異なり，かつレジスタの依存性がない場合に限り同時発行できる．

2命令間が同時発行できないグループにあるとき，実行に必要なハードウェア資源が競合するとき，レジスタの依存関係があるときは同時発行できない．この場合，第2命令は，その後続命令とともに再度組み合わされて，同時発行ができるか否かを決定する．この関係を**図11**に示す．

マニュアルを読み間違えていなければ，マルチステップ命令(この種の命令はハードウェア資源を独り占めする)が出現する場合や，レジスタの依存関係がある場合は，SH-4のパイプライン処理の効率はシングルパイプラインと大差ないように思える．少しの並列実行を行うために，ハードウェア資源をむだ使いしているように感じるのは筆者だけであろうか．せめて，整数演算器がもう一つあれば，性能はもう少し向上することだろう．

● 特徴的なSH-4のパイプライン動作例

ここで，SH-4のパイプラインで特徴的な二つの例を示す．

▶0サイクルレイテンシ命令はレジスタの依存関係があってもストールしない

これはソースフォワード機能と呼ばれる(通常のフォワードとは異なる)．おそらく，デコード時に他方の命令のソースオペランドを読み替えるのであろう．たとえば，

```
MOV   R1, R0   // R0 <- R1
ADD   R0, R2   // R2 <- R0+R2
```

という命令列がある場合，ADDのソースであるR0はR1に変換されて，依存性をなくし，この2命令を1サイクルで実行する．

▶条件分岐命令でオフセットが0の場合は分岐成立時でも1サイクルで実行できる

分岐命令が分岐する場合，分岐先のフェッチまでに1サイクルストールするので，レイテンシは2であるが，分岐のオフセットが0(2命令後への分岐)の場合は，分岐先フェッチのオーバヘッドがなく，1サイクルで実行できる．つまり，スキップ命令として利用できる．

たとえば，次のような命令列が考えられる．最近のMPUでも，2命令後への分岐は特別扱いして高速化が試みられているが，これはその先駆けかもしれない



I	：命令フェッチ
D	：命令デコード 発行 レジスタリード PC演算（一般命令）
EX	：演算，アドレス計算（ロード/ストア）
SX	：演算，アドレス計算（ロード/ストア）
MA	：メモリデータアクセス
S	：ライトバック
NA	：非メモリデータアクセス
F0/F1/F2/F3	：計算
FS	：計算，ライトバック

図10　SH-4のパイプラインの命令処理パターン

(しかし，そのような場合は条件MOVE命令を使用することのほうが多いような気もするが)．

```
CMP/GT R1, R2   // if R2>R1 then
                   set T-bit
BT     label    // branch relative if
                   T-bit set
ADD    #4, R3   // if T-bit set then
                   R3<-R3+4
label:
```

SH-4のパイプラインは2命令同時実行に制限が多い．このあたりはコンパイラの頑張りしだいといえるかもしれない．

ルネサステクノロジ(当時は日立製作所)の発表では，SH-4の性能は200MHz動作で360MIPSということになっている．これはDhrystone1.1による値であり，他社が採用しているDhrystone2.1では300MIPS程度という噂もある(公式発表はない)．200MHzで300MIPSも達成できれば，インオーダスーパースカラとしては，まずまずの性能であろう．しかし，300MIPSというのも少し眉唾な気がしないでもないが(このパイプライン構造の割に性能が良すぎる)．

● SH-5ではシングルパイプラインへ

SH-4の後継機種である**SH-5**では，スーパースカラ

SHAD R0, R1	I	D	EX	NA	S		
ADD R2, R3	I		D	EX	NA	S	
NEXT		I	D	EX	NA	S	

EXグループのSHADと同じEXグループのADDは並列実行できない．SHADのみ発行され，ADDは次の命令と組み合わされて並列実行可能か否かを調べる

（a）直列実行：並列実行不可能

ADD R2, R1	I	D	EX	NA	S
MOV.L @R4, R5	I	D	EX	MA	S

EXグループのADDとLSグループのMOVは並列実行できる

（b）並列実行可能：並列実行可能かつ依存性なし

ADD R2, R1	I	D	EX	NA	S	
MOV.L @R1, R2	I		D	EX	NA	S
NEXT		I	D	EX	NA	S

EXグループのADDとLSグループのMOVは並列実行可能．しかし，ADDの結果をアドレスとするので依存性があり，この場合は直列実行になる．MOVは次の命令と組み合わされて並列実行可能か否かを調べる

（c）並列実行可能：並列実行可能かつ依存性あり

MOV R0, R1	I	D	EX	NA	S
ADD R2, R1	I	D	EX	NA	S

依存関係がある場合でも，0サイクルレイテンシ命令の後続命令は並列実行できる．R1がフォワーディングされる

（d）0サイクルレイテンシ命令

MOV.L @R1, R2	I	D	EX	MA	S		
ADD R0, R2	I	D			EX	NA	S

MOV.Lは2サイクルレイテンシであり，その結果を後続命令で参照するには，MAステージまでストールする

（e）2サイクルレイテンシ命令

MOV.L @R1, R2	I	D	EX	MA	S			
SHAD R2, R3	I	D			d	EX	NA	S

ロード命令の結果をシフト量として使用する場合は，Sステージまでストールする．シフト量にはフォワーディングされる経路がない？

（f）シフト命令のシフト量

図11　SH-4の並列動作の組み合わせ

構造では制御が複雑で動作周波数を上げることができないという理由で，7ステージのシングルパイプラインが採用された．これで，動作周波数はSH-4の200MHzから400MHzへと向上するという．しかし，筆者にはSH-4のパイプラインが周波数の向上の妨げになるほど複雑だとは思えない(とはいえ規模は大きそうだが)．むしろ，SH-4ではシングルパイプライン並みの効率のスーパースカラでしかなかったので，効率的なシングルパイプラインで作り直したと考えるほうが納得できる．

実際，SH-5の性能は400MHz動作時に714MIPS(Dhrystone1.1)と発表されており，IPCでみるとSH-4もSH-5もほとんど同性能と思われる．これが本当なら，SH-5はとても素晴らしいCPUといえる．個人的にはシングルパイプラインでMIPS/MHzの値が1.785というのは不可能だと思うのだが(どう考えても，インオーダスーパースカラの値)．その後，SH-5の性能はDhrystone2.1では604MIPSと発表された．MIPS/MHzは1.51である．これでも性能が良すぎる気がする．こちらもSH-4の性能比(噂であるが)と同じである．

なお，SuperH社は，SH-6では再びスーパースカラ構成になると発表している．結局は，シングルパイプラインでは性能的に見劣りがするということか．

7 SH-X

● SH4-DSPの進化形？

業界では，SH3-DSPの後継としてSH4-DSPが出ると2000年頃から噂されていた．しかし，SH-4をCPUコアとすると消費電力が高くなるせいか，なかなか世

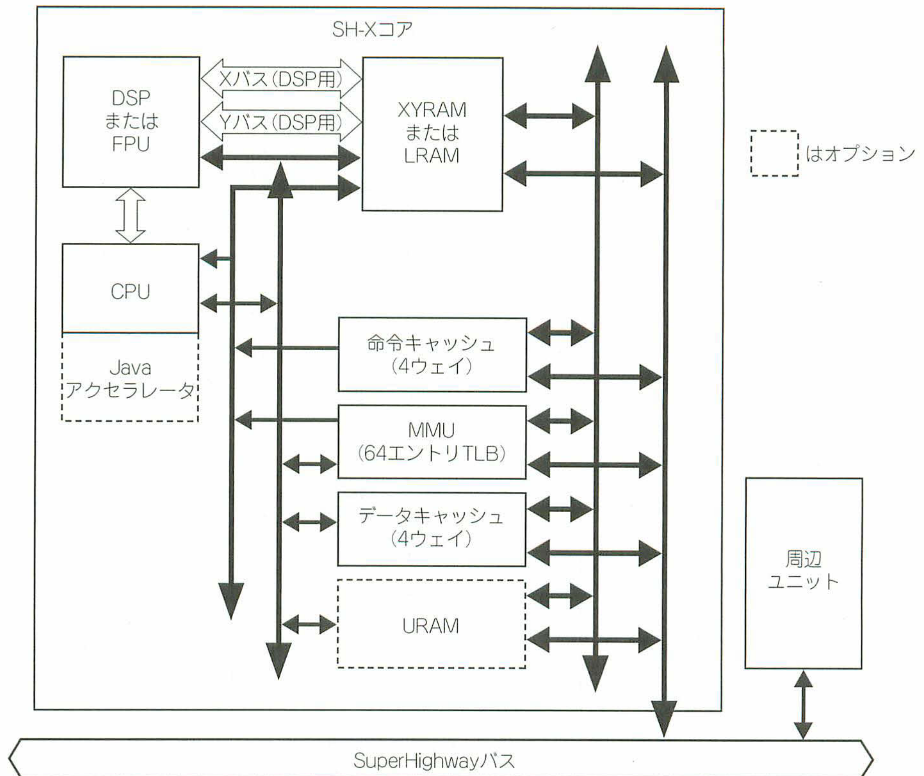


図12
SH-Xのブロック図

の中に出てこなかった．そうこうするうちに，2003年10月のMicroprocessor ForumでSH-Xが発表された．その仕様は，まさにSH4-DSPといえるものであり，高性能と低消費電力を達成するためにさまざまな工夫が施されている．

● 驚異的な性能

命令キャッシュとデータキャッシュのステージをそれぞれ1段増加させて，キャッシュアクセスに余裕をもたせた．このため，SH-3やSH-5の5段パイプラインから2段増えて7段パイプラインになった．これにより，0.13 μmプロセスで，400MHz動作を達成する．もっとも，ルネサステクノロジはSH-XでSH3-DSPの置き換えをねらっている節があり(応用分野は携帯機器？)，216MHzを現実的な動作周波数と想定しているようである．

消費電力は0.4mW/MHz(キャッシュを含む)で，400MHz動作時の消費電力は160mWと驚異的に小さい．基本的には，ゲーティッドクロックとキャッシュのバンク分割を採用しているだけだが，何か秘密がありそうである．特徴的な技術として，キャッシュのデータRAMはタグがヒットしたブロックしか活性化しない．この効果が大きいのだろうか．

また，SH-4とSH3-DSPを融合して進化させたスーパースカラ構造を採用し，性能は1.8DhrystoneMIPS/MHzを達成する．つまり，400MHz動作時には720MIPSと，これまた驚異的である．さらに売りが，720MIPS/0.16mW = 4500MIPS/W@1.0Vのパフォーマンスである．まあ，いくらスーパースカラの性能がよくても，1.8DhrystoneMIPS/MHzは大袈裟(誇大広告)と思うが，ルネサステクノロジの自信(強気)が窺える．

● SH-Xの構成とのパイプライン

図12にSH-Xのブロック図を示す．SH-4とSH3-DSPとの融合ということで，DSPとその作業領域であるXYRAMを内蔵する．

図13にSH-Xのパイプラインを示す．SH-4とは異なり整数ユニットを2個備えた2ウェイスーパースカラである．整数パイプラインと，DSPパイプライン，FPUパイプラインは並行動作可能である．SH-Xの構成としては，DSPやFPUなし，DSPあり，FPUあり，の3種類が可能である．

キャッシュアクセスに余裕をもたせるために，パイプラインを，従来の5段から7段に変更した．しかし，パイプラインを7段にしたことで，分岐ペナルティとロードユース(ロードの結果を待ち合わせるインター

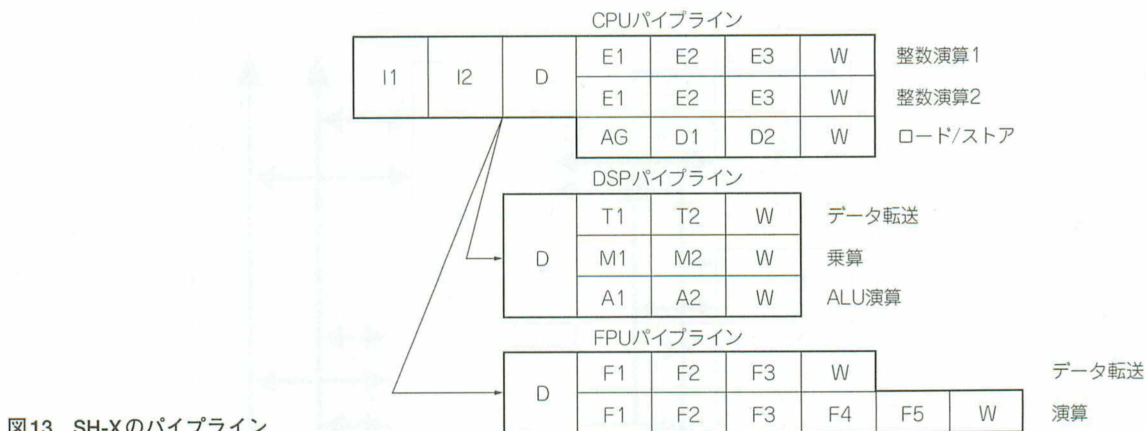


図13　SH-Xのパイプライン

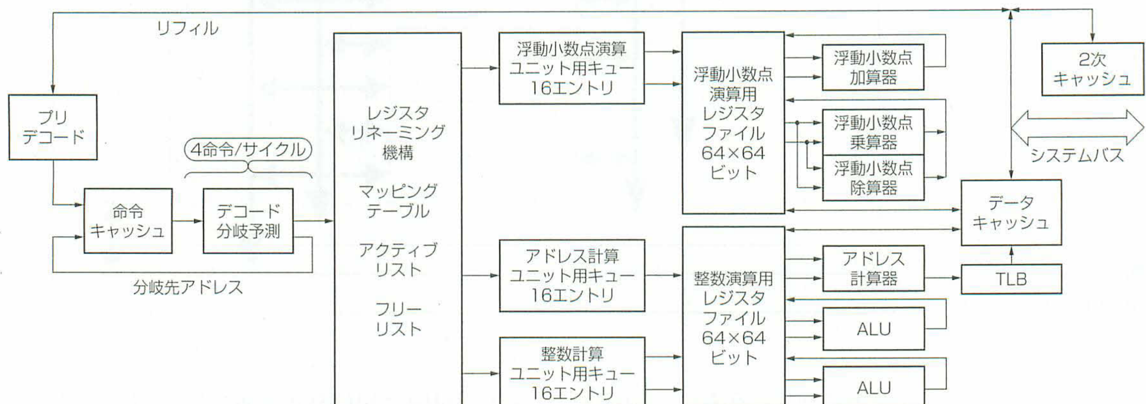


図14　R10000のブロック図

ロック）により，設計当初，同一クロックで36％の性能低下を招いたという．これを削減するために，分岐予測の強化（30％改善）とディレイドALU方式の採用（6％改善）を行い，SH-4並みのパイプライン効率を達成している．そのため，性能はSH-4と同じ1.8DhrystoneMIPS/MHzである（SH-4はDhrystone1.1，SH-XはDhrystone2.1の性能であるが，細かいことは言わないのが業界人の情け？）．

ディレイドALU（Delayed ALU）とは，整数演算のALUへのデータ供給をできるだけ後にずらす（SH-Xの場合は1クロック）ことにより，ALUでの待ち合わせを低減する手法である．スキュードALU（Skewed ALU）という方式もこれと同じものであろう．

8 R10000

● 4ウェイスーパースカラ構造

次はMIPS系のプロセッサから，**R10000**を取り上げてみる．**図14**にR10000の機能ブロック図を示す．

図を見てわかるようにR10000は，演算器として2個の整数ALU，アドレス計算（ロード/ストア）ユニット，浮動小数点加算器，浮動小数点乗算器，浮動小数点除算器（平方根器を兼ねる）をもつ．各演算器には，それぞれ16エントリからなる整数キュー，アドレスキュー，浮動小数点キューが接続されている．命令は各キューから対応する演算器に対してアウトオブオーダで発行される．

R10000は1サイクルで，4命令を命令キャッシュからリードし，最大4命令をデコード/レジスタリネーム可能である．デコードされた命令は32エントリのアクティブリストと呼ばれる命令キューと兼用のリオーダバッファに格納され，インオーダなリタイア〔R10000ではグラジュエート（卒業）と呼ぶ〕を管理する．1サイクルで最大4命令をリタイアできる．

演算器を6個備えているが，浮動小数点の乗算器と除算器の入力が共通なので，1度に最大5命令を発行することができる．その意味で5ウェイのスーパースカラであるが，デコード自体は1サイクルで最大4命

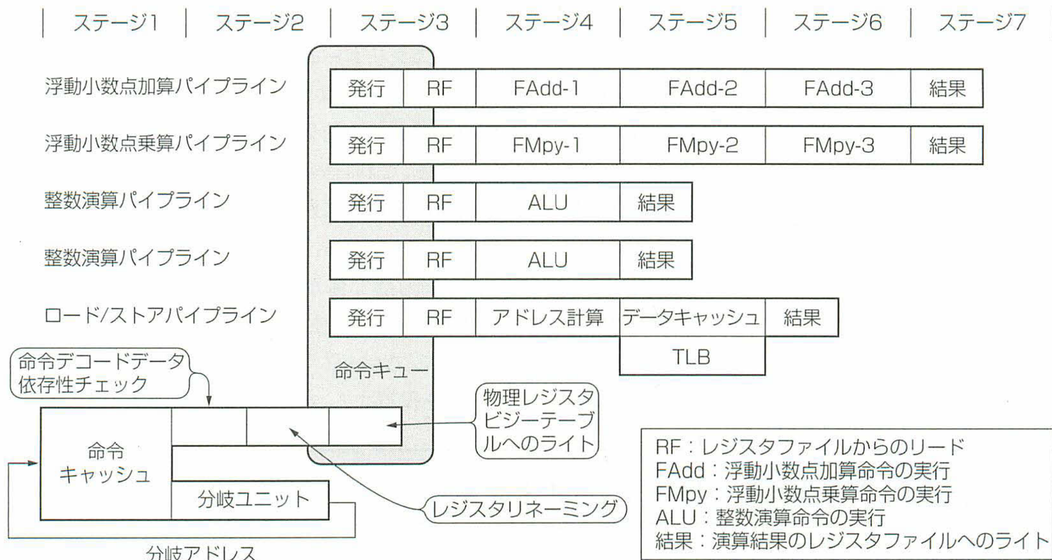


図15　R10000のパイプライン

令なので一般には4ウェイのスーパースカラと呼ばれている．

図15にR10000のパイプラインを示す．一応，7ステージパイプラインの体裁をとっているが，命令のデコード/レジスタリネーム/キューへの格納(ステージ1，ステージ2)までと，発行/実行/結果の格納(ステージ3～ステージ7)までのステージは独立に動作する分離(decoupled)方式である．アウトオブオーダ方式のスーパースカラとしては珍しくない．

R10000のアーキテクチャをMIPS社は**ANDES** (Architecture with Non-Sequential Dynamic Execution Scheduling)と呼び，アウトオブオーダ実行，分岐予測，投機実行などの総称としている．レジスタリネームは32個の論理レジスタを64個の物理レジスタへ割り当てる．割り当て可能な物理レジスタはフリーリストと呼ばれるテーブルで管理される．物理レジスタは整数と浮動小数点の2系統が用意されている．

分岐予測は，512エントリ×2ビットの情報で過去2回の分岐/不分岐の履歴を保持し，予測した方向に投機実行する．投機実行は，アクティブリストが一杯になるか，レジスタリネームのためのフリーリストが空になるまで，あるいは分岐命令を4個デコードするまで(つまり4回続けて分岐予測するまで)継続される．分岐予測のたびに，その時点での論理レジスタと物理レジスタの対応表(マップテーブル)のコピーを残しておき，分岐予測が失敗すると，そのコピーに基づいてパイプラインを分岐元の状態に戻す．R10000では演算はすべて物理レジスタに対して行われ，アーキテクチャ上のレジスタ(論理レジスタ)にはリタイア時に対応する物理レジスタから書き戻される．

なお，整数キューとアドレスキューは，基本的には，整数レジスタが対象なので，2種類持つ必然性は少ない．しかし，ノンブロッキングキャッシュの実現を容易にするためにアドレスキューが特設されているものと推測される．もっとも，MIPSの命令セットにはデスティネーションレジスタが浮動小数点レジスタのものもあるので，そのせいかもしれない．

● 命令フェッチ

ここで，R10000のパイプラインの詳細について説明する．

各サイクルにおいて，R10000は32Kバイトの2ウェイセットアソシアティブ構成の命令キャッシュから4命令をフェッチできる．命令キャッシュに格納されている命令は37ビット長で，命令キャッシュのリフィル時に32ビット長の命令をプリデコードしたものである．この余分な5ビットによって各命令を分類し，その命令が実行されるユニット情報を付加することで命令デコードを効率的に行える．このプリフェッチ/プリデコードステージはパイプラインステージの中には数えられていない．

余談であるが，一つの命令長である32ビット(4バイト)が37ビットに拡張されて格納されるので，命令キャッシュの容量は正確には37Kバイトである．

● 命令デコード

命令キャッシュからフェッチされた命令は，レイテンシが2ステージの命令デコーダに渡される．実際のデコードは最初のステージで行われ，2番目のステージではレジスタリネームが行われる．

MIPSアーキテクチャではレジスタの本数は整数用と浮動小数点用にそれぞれ32本ある．これは論理レジスタと呼ばれる．R10000はさらにレジスタリネーム用に整数用と浮動小数点用の物理レジスタをそれぞれ64本備えている．プログラム的には32本の(論理)レジスタしか見えてないが，MPU内部では2倍の本数の(物理)レジスタが処理結果を保持している．

レジスタリネームはアウトオブオーダの投機実行を実現するために重要である．R10000は演算の中間結果や投機実行の結果を，この不可視な物理レジスタに保持する．これらの結果はすべての依存性が解決され，投機的な実行経路が消え去ったときにプログラムから見えるようになる．

MPU内部で何が起きているかを管理するために，R10000はすでに使用されているレジスタ(の番号)を保持するアクティブリストと利用可能なレジスタ(の番号)を保持するフリーリストを用意している．アク

Column 2 V_R4131のパイプライン

V_R4131はNECが開発した64ビットMIPSプロセッサである．同社の組み込み制御用MIPS RISCとしては初めてスーパースカラ構造を採用した．V_R4131の特徴は，性能もさることながら，低消費電力である点である．通常のスーパースカラ構造では消費電力が増大するため，ユニークなパイプライン構造を採用している．V_R4131では$8n$番地と$(8n+4)$番地の命令が必ず組で処理される．このため，スーパースカラと呼ぶよりもVLIWと呼ぶほうがすっきりする．VLIW構造を採用することで制御回路を単純化し，低消費電力を図っているのである．これは，Transmeta社のCrusoeの主張に近い．

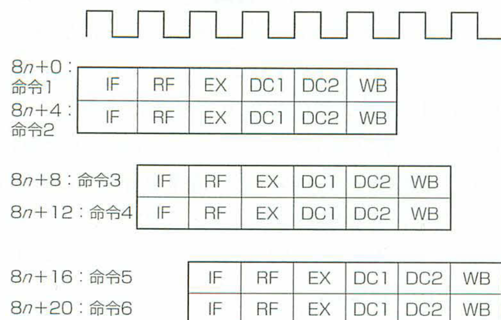
V_R4131のパイプラインは典型的な次の6ステージからなる．

1) IF ：命令フェッチ
2) RF ：レジスタフェッチ/命令デコード
3) EX ：実行
4) DC1 ：データキャッシュ(その1)
5) DC2 ：データキャッシュ(その2)
6) WB ：ライトバック

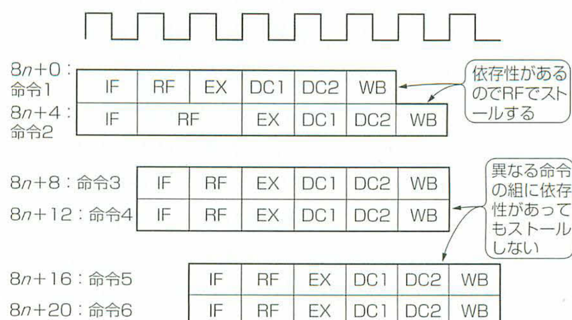
このパイプラインを一言で表せば「隣接する2命令単位で処理するシングルパイプライン」である．MIPSの命令は4バイト固定長なので，$8n$番地と$(8n+4)$番地の命令を一つの命令とみなし，それを6ステージのシングルパイプラインで実行する〔図A(a)〕．そのためにすべての演算器を2系統備えている(乗除算とロード/ストアユニットは例外)．

隣接する2命令を同時に実行するわけであるが，その命令間にレジスタのRAW依存性がある場合は，当然ながら不都合が生じる．それを回避するために，V_R4131ではレジスタのRAW依存がある場合は，$(8n+4)$番地の命令をRFステージで1クロック待ち合わせて，$8n$番地の演算結果をEXステージからフォワーディングする．つまり，2命令を依存性がない場合は1クロックかけて実行し，依存性がある場合は2クロックかけて実行する〔図A(b)〕．また，実行ユニットを一つしか備えない乗除算命令やロード/ストア命令が隣接する場合は，強制的に依存関係を発生させて逐次的に実行させると思われる．

図A V_R4131のパイプライン



(a) V_R4131のパイプライン（依存性なし）



(b) V_R4131のパイプライン（依存性あり）

ティブリスト内のレジスタは二つの状態を取ることができる．一つは「アクティブ」である．つまり，実行中の命令で使用されている状態である．もう一つは「コンプリート」である．つまり，命令実行の最終結果を示す状態である．

ある瞬間には最大32命令が「アクティブ」状態にある．「コンプリート」状態になり結果がグラジュエートすると，不要になったレジスタはアクティブリストから削除されフリーリストに返却される．投機実行はフリーリストに利用可能なレジスタがあり，レジスタリネームが可能な限り継続できる(アクティブリストに空きがあることも必要)．

レジスタリネームは，また，分岐予測において重要な役割を果たす．つまり，分岐が誤って予測されたときに，投機的な実行経路を迅速に破棄する役割がある．R10000は分岐ごとに，最大四つの分岐の深さまで，投機実行の入れ子が可能である．実行経路の分岐点ごとに，レジスタ状態のコピーをもつ．そのコピーは，MIPSの表現によると，その時点で存在しているレジスタリネームマップ(割り当て表)のシャドウマップと呼ばれる．後に分岐予測が失敗したことが判明しても，R10000はバッファのフラッシュやレジスタのクリアを行う必要はない．単に(予測を誤った分岐命令に対応する)最適なシャドウマップを現状のレジスタリネームマップにコピーし直すだけでよい．そのとき，無効な結果を保持しているレジスタはフリーリストへ返却される．この操作は1サイクルで行われる．このため，誤った分岐予測のペナルティは1～4サイクルになる．これはR10000が何回誤った予測を行ったかに依存する．最大の入れ子である4個の分岐予測を誤った場合が最悪の4サイクルになる．

分岐予測も動的に行われる．R10000は2ビットの情報で各分岐の履歴を記憶しておく．これは4種類の状態，つまり，「強く分岐する」，「弱く分岐する」，「弱く分岐しない」，「強く分岐しない」である．MIPSによるとR10000の分岐予測の正確さは90％以上であるという．これは，典型的な動的な命令列において，平均的に6命令に1回分岐が現われるという根拠に基づいているらしい．

● 命令実行

R10000は各サイクルにおいて，最大4命令をフェッチし，最大4命令をグラジュエートするが，その中間には五つの実行ユニットがある．このため，可能性としては，各サイクルで，5命令を同時発行，実行，コンプリートできる．このため，R10000は4ウェイスーパースカラとも5ウェイスーパースカラとも呼ばれる．しかし，命令の処理数に関するこの不整合は偶然ではない．ピークのバンド幅を大きくしておくことで，内部資源の割り当てが効率よく行え，将来の拡張の余地も残している．……という説明はもっともらしいが，本当にピーク時のバンド幅を考慮するなら，整数演算ALUもFPUも4個ずつ用意すべきであろう．実際，後継機種ではそのような構成を採るという動きもあったようだが，いまだ実現には至っていない．

機能ユニットは二つの64ビット整数演算ALU，一つのロード/ストアユニット，二つのFPUからなる．FPUのうち，一つは加減算用，残りは乗除算/平方根用である．後者のFPUは実際には，同一の発行/コンプリート論理を共有する，乗算，除算(平方根を含む)を行うサブユニットの組である．それらは浮動小数点の乗算と除算を(同時発行はできないが)並行に実行できる．

二つの64ビットALUはほとんど同一である．ただし，乗除算は一方のALUでしか処理できない．他方のユニットには分岐予測の結果を確かめる論理がある．シフトも一方のユニットでしか実行できない．ロード/ストアユニットはすべてのアドレス計算，アドレス変換を処理する．

ここで問題となるのは`NOP`命令である．MIPSアーキテクチャの`NOP`命令は「`SLL r0, r0, r0`」，つまりシフト命令である．プログラム中にかなりの頻度で出現する`NOP`命令が片方のALUでしか実行できないというのは性能上も問題である．これを回避するため，R10000では`NOP`命令はプリデコード時に「`ADD r0, r0, r0`」などの並列実行可能な命令に変換していると聞く．また，これを考慮してか，最新のMIPS32/MIPS64アーキテクチャでは`SSNOP`(SuperScalar NOP)なる命令が定義されている．

五つの実行パイプラインはすべて，最低1ステージからなる実行ステージと，上述のフェッチ，デコード，リネームステージをもち，最後がグラジュエートステージである．このためパイプラインの最小ステージ数は5ステージである．

命令は最初の3ステージを通過するときにはプログラムの順序を維持している．そして，3種類のキューに格納され，最適な実行ユニットに発行されるのを待つ．これらのキュー(ALU，FPU，ロード/ストアユニット用)のそれぞれは16エントリからなり，そのキ

ューのどの位置からでも発行ができる．つまり，この時点からプログラムの順序を維持しなくなる．

ある条件下では，R10000は1サイクルで最大5命令をキューからアウトオブオーダに発行できる．しかし，多くの場合は，命令の依存性に応じて1～4命令を発行する．IPCから察するに，平均は2命令前後であろう．

ロード処理がデータキャッシュにヒットするときには2サイクルかかる．また，ロードは投機的にアウトオブオーダで実行される．これに加え，ノンブロッキングキャッシュ構造によりロードを効率的に処理する．ノンブロッキングとは，ロードがキャッシュにミスしてもストールすることなく先に進める技術である．アウトオブオーダで命令の追い越しが可能な場合は，とくに効果的である．R10000では最大四つのロードをノンブロッキングで実行できる．

● グラジュエート（リタイア）

グラジュエートとは物理レジスタの内容を対応する論理レジスタに書き戻す処理である．リタイアとも呼ばれ，インオーダな完了を実現する．

パイプラインの最終ステージにおいて，命令がコンプリートしていても，すべての依存性が解決され，投機的な実行経路が確定するまでは，グラジュエートできない．R10000は正確な例外を保証するので，例外を起こす命令の後続命令はコンプリートしていても，その命令は同様にグラジュエートできない．

グラジュエート時には，物理レジスタが論理レジスタにリネームし直され，その内容が有効になる．このとき，もっとも前にコンプリートした命令から最初にグラジュエートする．グラジュエートを管理するのはアクティブリストである．あるサイクルにおいて，アクティブリストの先頭から見て連続してコンプリートしている命令が，その時点でのグラジュエートの対象になる．したがって，アクティブリストの先頭の命令がコンプリートしない限りは1命令もグラジュエートできない．R10000は各サイクルで，最大4命令をグラジュエートできる．この操作により，命令の流れがその本来のプログラムの順序に戻される．

● R10000の性能

MIPS社の発表ではR10000のIPCは1.5だという．これが，Dhrystone MIPSではなく，本来の意味のMIPS値から求められたものとすればかなりの高性能である．出典は失念したが，4ウェイスーパースカラではIPCは1.5程度が限界だそうである．つまり，R10000はIPC的には究極の性能を達成しているといえなくもない．

余談だが，R10000の後継機種である**R12000**では，動作周波数が200MHzから300MHzに引き上げられたほかに，マイクロアーキテクチャ的には，アクティブリストが48エントリに，分岐予測テーブルが2048エントリに増加している．また新たに32エントリの分岐ターゲットバッファが追加されていた．R12000の後継機種として**R14000**，**R14000A**，**R16000**が開発されており，その動作周波数は500MHz，600MHz，700MHzである．とくにR16000は2003年4月に発表されたが，1GHzをはるかに下回る周波数は寂しいものがある．

R18000は2001年のHotChipsシンポジウムで概要が発表された．仮想アドレス空間の拡張（52ビット），2個のFPUを実装，L2キャッシュ（1Mバイト）の内蔵とL3キャッシュ（最大64Mバイト）インターフェースの内蔵という点が新たに公開された．動作周波数は800MHz～1GHzと予想されている．周波数的には時代遅れの感がなきにしもあらずである．

9 PowerPC750

● パイプライン

次はPowerPCアーキテクチャのプロセッサを取り上げてみよう．PowerPCはIBMとMotorolaが独自に開発しているMPUであり，その種類も豊富である．しかし，その内部構造はどれも似ている．

ここでは，任天堂のGAMECUBEのMPUであるGekkoの基になった**PowerPC750**について説明する．GekkoはIBMが開発したMPUで，PowerPC750（以降PPC750と略）を下敷きに単精度FPUの強化とL2キャッシュを内蔵したものである．

図16にPPC750の内部ブロック図を示す．また，命令の流れとパイプラインは，それぞれ，**図17**，**図18**のようになる．以下にそれぞれのユニットの動作について説明する．なお，**図18**の各ステージの説明は次のとおりである．

▶ フェッチ

フェッチステージとは命令が要求されてから命令キューに格納されるまでの間を示す．レイテンシは可変で，命令がBTIC，内蔵キャッシュ，L2キャッシュ，システムメモリのどこにあるかに依存する．命令キューはIQ0～IQ5の六つのエントリをもっている．

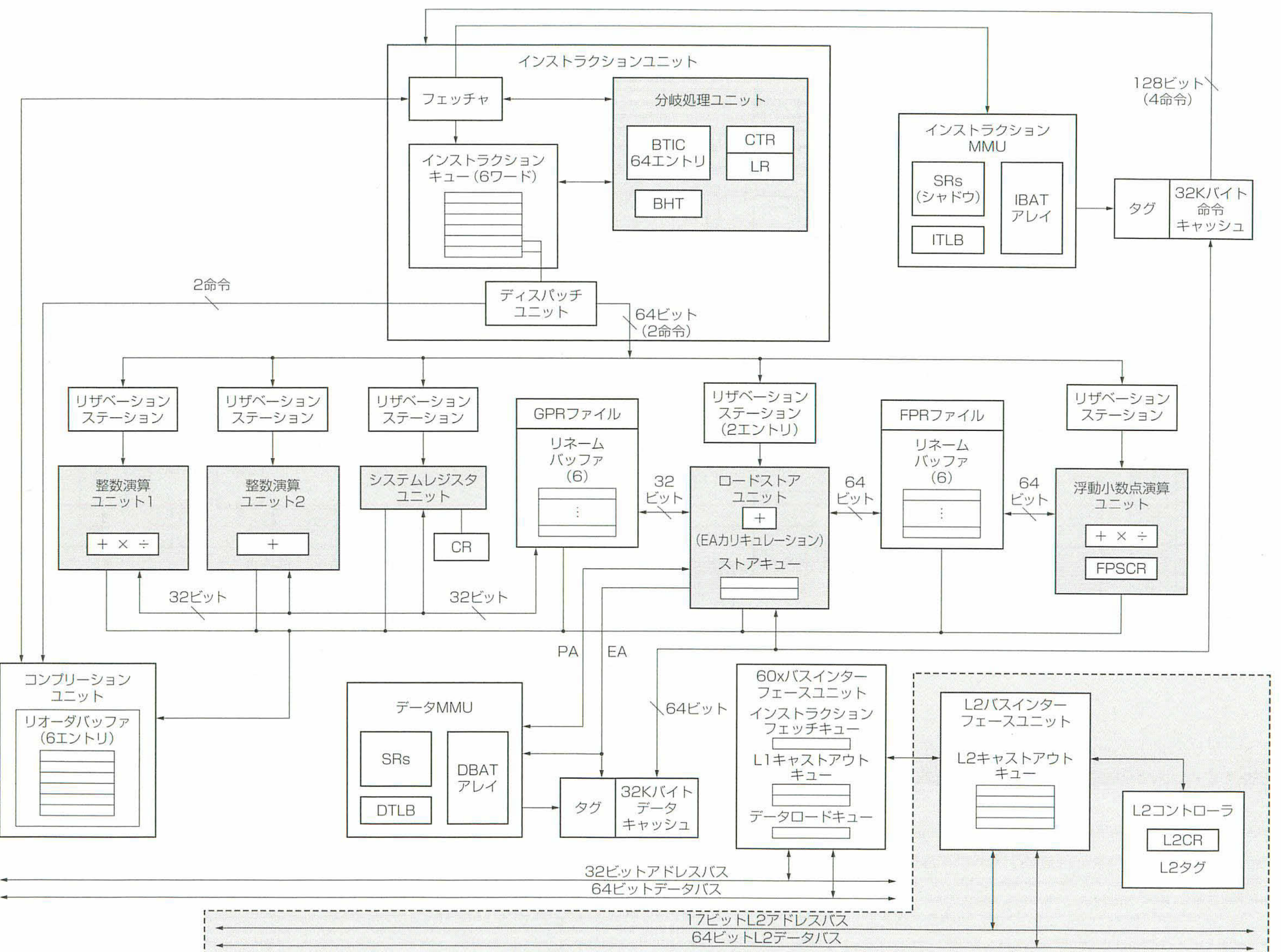


図16　PowerPC750のブロック図

フェッチ

インストラクションキュー

IQ5 → IQ4 → IQ3 → IQ2 → IQ1 → IQ0

ブランチ
プロセッシング
ユニット

ディスパッチ

コンプリ
ーション
キュー

リザベーシ
ョンステー
ション

FPU

LSU

IU1

IU2

SRU

ストアキュー

IQ5 → IQ4 → IQ3 → IQ2 → IQ1 → IQ0

コンプリート
(リタイア)

コンプリーション
キュー

図17　PowerPC750のデータの流れ

フェッチ

命令
キャッシュ

分岐予測

命令キュー

IU1，IU2，SRU，LSUには1サイクルに2命令発行可能
BPUは独立に発行可能

ディスパッチ
エントリ

「完了キュー」ステージのサイクル数は0以上

IU1パイプライン | 実行 | 完了キュー | 完了/リタイア

IU1は整数乗除算器を備え，乗除算器では実行に複数サイクルかかる

IU2パイプライン | 実行 | 完了キュー | 完了/リタイア

SRUパイプライン | 実行 | 完了キュー | 完了/リタイア

LSUパイプライン | アドレス計算 | データキャッシュ | 位置合わせ | 完了キュー | 完了/リタイア

「データキャッシュ」ステージのサイクル数はヒット/ミスに依存する

FPUパイプライン | 乗算 | 加算 | 丸め/正規化 | 完了キュー | 完了/リタイア

BPUパイプライン | 完了キュー | 完了/リタイア

完了キューに入るのはLRやCTRを更新する分岐命令のみ

図18
PowerPC750の
パイプライン

▶ディスパッチエントリ中

命令はIQ0とIQ1からディスパッチされる．ディスパッチは瞬時に行われるので「フェッチステージの最後のサイクル」と「実行ステージの最初のサイクル」の中間の時点を表す事象として記述する．

▶実行

命令によって規定される処理はそれに最適な実行ユニットで行われる．この「実行ステージ」から完了（コンプリーション）キューに入る．

▶コンプリート

命令はコンプリーションキューにある．PPC750ではコンプリーションキューがリオーダバッファの役割をしている．最終ステージにおいて，実行された命令の結果がライトバックされ，命令はリタイアする．コンプリーションキューはCQ0～CQ5の六つのエントリをもっている．

▶リタイアメントエントリ中

コンプリートした命令はCQ0とCQ1からリタイアできる．ディスパッチと同様に，リタイアはコンプリートステージにおける最後のサイクルの終わりで発生する事象である．

● 命令の流れ

PowerPCにおいて命令は，命令ユニットから実行ユニットに流れて処理される．命令ユニットは逐次フェッチ器，6エントリの命令キュー（IQ），ディスパッチユニット，分岐処理ユニット（BPU）からなる．逐次フェッチ器とBPUから供給される情報に基づいてフェッチすべき次のアドレスが決定される．そして，逐次フェッチ器は命令キャッシュから命令を読み出して命令キューに渡す．BPUは逐次フェッチ器に読み込まれる分岐命令を解析し，分岐条件が確定している場合は分岐命令を削除し，フェッチを分岐先に切り替える．分岐条件が確定していない場合は，静的または動的な分岐予測を行い，投機実行をする．分岐条件が確定する間にフェッチした命令はBPUに保持される．

図16や**図17**に示すように，PPC750では命令キューとリザベーションステーションが分離されている．命令キューが，いわゆる通常のリザベーションステーションに相当し，PPC750のリザベーションステーションは，後続命令を時間的なロスなしに実行するための，単なるバッファの役割しかしていない（ように思える）．

命令はインオーダに各実行ユニットのリザベーションステーションに渡されて逐次的に実行される．各実行ユニットの処理は独立しているで，全体としてはアウトオブオーダ実行になる．命令はコンプリーションキュー（コンプリーションユニットのリオーダバッファのこと）の管理の下，インオーダに終了する．

命令の処理はリネームされたレジスタに対して行われ，コンプリーションユニットによるリタイア時にアーキテクチャ上のレジスタ（GPR，FPR）に書き戻される．

● 命令キューとディスパッチユニット

命令キュー（IQ）は最大6命令を格納できる．命令フェッチ器はIQに空きができると直ちに命令キャッシュから命令を読み込む．そして，分岐命令を除くすべての命令は，1クロックに最大2命令の割合で，IQの先頭とその次（IQ0，IQ1）から各実行ユニットにディスパッチされる．五つの実行ユニット（IU1，IU2，FPU，LSU，SRU）はそれぞれ独立にリザベーションステーションをもっている．ディスパッチユニットはソースとデスティネーションレジスタの依存性をチェックし，コンプリーションキューに空きがあれば新しい命令を，その命令が実行されるべき実行ユニットにディスパッチする．これは，各ユニットのリザベーションステーションに命令を移動することに等しい．

● 分岐処理ユニット（BPU）

BPUは，逐次フェッチ器から分岐命令を受け取り，条件分岐に関するCR（Condition Register）の先読み操作を行うことで分岐条件を早期に確定する．これにより，多くの場合は0サイクルで分岐を実現できる．無条件分岐と条件が確定している条件分岐は直ちに実行される．

分岐条件が未確定な条件分岐命令に関しては，アーキテクチャ的に定義された静的な分岐予測または動的な分岐予測を用いて分岐の経路が予測される．予測により，予測した経路から，命令フェッチ，ディスパッチ，実行が行われる．しかし，予測が正しいと決定（解決）されるまで，それまでに処理した命令はコンプリートすることができず，結果をレジスタにライトバックすることもできない．

もし予測を誤った場合，誤った経路の命令はプロセッサからフラッシュされ，正しい経路からの処理が開始される．いわゆる投機実行であるが，PPC750は2番目の分岐命令まで予測することができる．2番目に予測された命令の経路からは，命令のフェッチはできるがディスパッチはできない．

動的な分岐予測には512エントリの分岐履歴テーブル（BHT）を用いる．これは各エントリに2ビットの情

報を保持し，それらは，分岐命令に対する「NOT TAKEN」，「強いNOT TAKEN」，「TAKEN」，「強いTAKEN」という四つのレベルを示す．もし，動的な分岐予測ができない場合は，条件分岐命令に埋め込まれている予測ビットを使用する．

このように，PPC750は未解決の条件分岐命令に出会うと，分岐が解決するまで結果をアーキテクチャ上のレジスタにライトバックはしない（リネームレジスタに保持する）が，予測した分岐先からの経路からの命令を実行する（投機実行）．この実行は2番目の未解決な分岐命令に出会うまで継続する．もし，分岐がTAKEN（TAKENと予測）するとき，TAKENしない経路からの命令は破棄され，予測した側の命令の経路がIQにフェッチされる．

BTIC（Branch Target Instruction Cache）は，最近使用された分岐先の2命令を保持する64エントリのキャッシュである．命令フェッチがBTICにヒットするとき，命令は次のクロックサイクルで命令キューに与えられる．これは命令キャッシュからリードするよりも1クロック早い．それに引き続く命令は次のクロックサイクルで命令キャッシュからリードされる．つまり，BTICは誤って命令をディスパッチする回数を減らし，ターゲットストリームからの1クロック早い開始を可能にする．

BPUは分岐のターゲットアドレスを計算する加算器と3個のユーザーが制御するレジスタであるリンクレジスタ（LR），カウントレジスタ（CTR），CRを含む．BPUはある種の分岐命令ではサブルーチンコールの復帰位置を計算し，LRレジスタに格納する．さらにLRは分岐コンディショナルリンクレジスタ（bclrx）命令に対するターゲットアドレスを含む．CTRはカウントレジスタに対する条件分岐（bcctrx）命令の分岐ターゲットアドレスを含む．LRやCTRは特殊レジスタ（SPR）であり，その内容はGPR間で転送可能である．BPUはGPR（整数レジスタ）やFPR（浮動小数点レジスタ）と独立の特殊レジスタを使用するため，分岐命令の実行は整数や浮動小数点命令の実行とは別に，独立して行うことができる．

● コンプリーションユニット

コンプリーションユニットは命令ユニットと密接に連動して動作する．命令はプログラムの順序でフェッチされディスパッチされる．ディスパッチ時に，6エントリのコンプリーションキューの連続するエントリに，その命令を入れることによってプログラムの順序を保持する．コンプリーションユニットは，命令をディスパッチから実行を通じて追跡し，コンプリーションキューの二つの出口（CQ0，CQ1）からプログラムの順序でリタイアさせる．コンプリーションキューに空きができるまで命令を実行ユニットにディスパッチすることはできない．

また，CTRやLRを更新しない分岐命令は命令ストリームから削除され，コンプリーションキューのエントリを占有することはない．CTRやLRを更新する命令は，それらが実行ユニットに発行されないことを除き，非分岐命令と同様にディスパッチされコンプリートする．

命令をコンプリートさせることはアーキテクチャ上のレジスタ（GPRやFPR，LR，CTR）に実行結果をライトバックすることである．インオーダなコンプリーションを行うことにより，PowerPCが分岐予測の誤りを回復するときや例外を発生する場合の正しい動作を保証する．命令がリタイアするとき，その命令はコンプリーションキューから削除される．

10 Power4のパイプライン

● Power4とは

Power4は，UntraSPARCに対抗するために，最新技術の粋を集めて開発した，PowerPCアーキテクチャを実装するEWS向けMPUである．PowerPCは64ビットアーキテクチャを定義しているが，ほとんどの製品は32ビットアーキテクチャを実装する．しかし，Powerシリーズでは Power3から64ビットアーキテクチャを採用している．

Power4では，高い動作周波数で動作させるために，PowerPCよりも命令の実行処理を単純化してある．つまり，通常4命令から構成されるグループを形成し

注1：この意味で，Power4は一般のPowerPCよりもIPCが低いと考えられる．しかし，IBMのPowerPCは，Power4以降は，すべてPower4の技術を採用していると報道されている．その技術が何であるか具体的には不明である．PowerPCのパイプラインが，Power4と同様に，グループ内のみでのアウトオブオーダ実行になるとしたら，IBMはIPCよりも高い周波数での実行を選択したことになる．

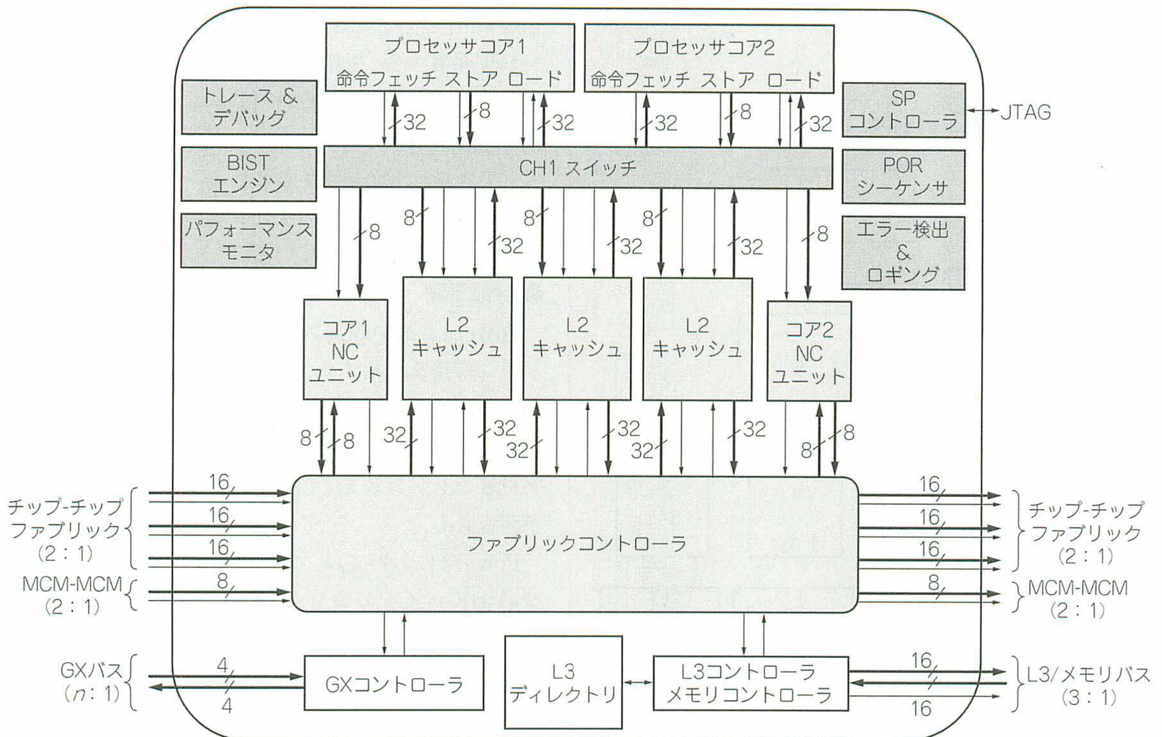


図19　Power4チップ

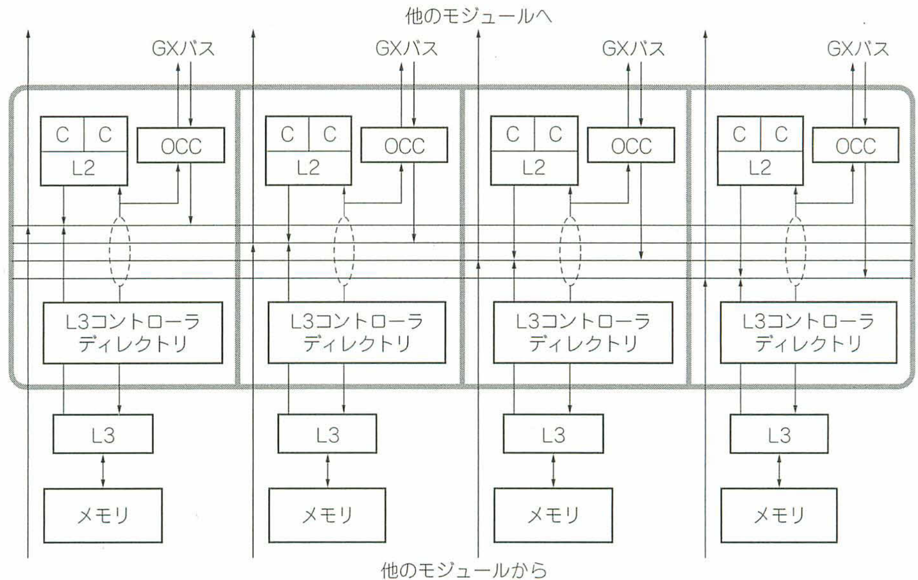


図20
Power4モジュール

て，グループを逐次的に実行する．グループ内の命令はアウトオブオーダで実行できるが，グループ内の全命令の実行が終了しないと，新しいグループを実行できない[注1]．

また，Power4は，最初から，SMP(Symmetric Multi-Processor)構成が可能なように設計されている．まず，Power4チップは二つのプロセッサを含むCMP(Chip Multi-Processor)である(**図19**)．これを4組SMP結合して，一つのMCM(Multi-Chip Module)を形成する(**図20**)．さらに，Power4のMCMは4枚の結合が可能で，最大32ウェイのCache Coherent NUMAシステム構成を可能とする．

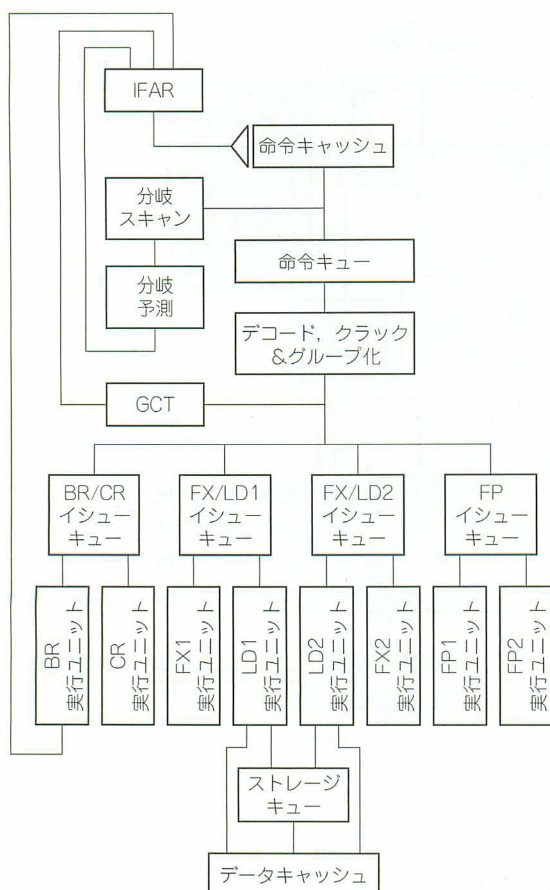


図21　Power4のブロック図

後年IBMは，Power4をシングルプロセッサ化し，AltiVec互換のSIMD命令を追加したPowerPC970を開発し，Power Macintosh G5用にリリースした．

以下では，単体のPower4プロセッサについて解説する．

● Power4プロセッサ

図21はPower4プロセッサのブロック図を示す．チップ上の二つのプロセッサは同一で，ソフトウェア制御のSMP構成を提供する．Power4のCPUコアのマイクロアーキテクチャは，投機実行可能なアウトオブオーダスーパスカラで，各サイクルに最大8命令を発行し，最大5命令を完了する．レジスタリネーミングプールと他のパイプライン構造をもつアウトオブオーダ動作可能な資源により，ある瞬間には200命令以上が「飛行中(in flight)」となる．命令レベルの並列性を引き出すために，毎クロックで命令発行可能な八つの実行ユニットを備える．さらに同機能のFPUを二つ備え，毎クロックで積和命令(fused multiply and add)を開始できる．つまり，各サイクルで，最大4個(乗算×2，加算×2)の浮動小数点命令を実行できる．二つの固定小数点(整数)ユニットは，効率的にデータを供給するために，それぞれに独立なロード/ストアユニットが結合されている．さらに，一つの分岐実行ユニット，一つの条件レジスタ関連の論理演算を行うユニットも備えている．

● 分岐予測

Power4も高い周波数を実現するため，長いパイプライン構造を採用する．このため，他のMPUと同様に分岐予測機構を備えている．Power4では，2段階の分岐予測機構を採用する．加えて，カウントレジスタやリンクレジスタで指定されるアドレスへの分岐も予測される．

Power4においては，最大8命令がダイレクトマップの64Kバイトの命令キャッシュから，各サイクルでフェッチされる．分岐予測機構はフェッチされた命令を走査し，各サイクルで最高二つの分岐命令を認識できる．分岐命令のタイプに応じ，いろいろな分岐予測機構が分岐の方向と分岐先アドレスを予測する．当然のことながら，無条件分岐に対する分岐方向は予測されない．無条件にフェッチ先が切り替わる．分岐予測対象は条件分岐であり，命令フェッチ時に条件レジスタのビットが判明していれば，直ちに分岐/非分岐を判断する．

リンクレジスタへのブランチ命令(bclr)とカウントレジスタへの分岐命令(bcctr)の分岐先アドレスは，それぞれ，ハードウェアで実装されたリンクスタックとカウントキャッシュを用いて予測される．絶対アドレスと相対アドレスによる分岐命令の分岐先は，分岐命令の走査時に計算される．

分岐命令はパイプラインを進んでいき，最終的には分岐実行ユニットで実行される．そこで，最終的な分岐方向を決定する．この時点で分岐予測が正しければ，分岐命令は単に他の命令と同様に終了する．分岐予測が間違っていれば，それまでにフェッチした命令を廃棄し，正しい方向からフェッチをし直す．典型的な投機実行である．

Power4では，分岐命令の方向を予測するために，3種類の分岐履歴テーブルを使用する．

第1のテーブルは局所予測器と呼ばれる．これは，伝統的な分岐履歴テーブル(BHT：branch-history table)と似ている．これは，分岐命令のアドレスによってインデクスされる16384(16K)エントリの配列で，

分岐が成立するかしないかの1ビットの情報を出力する．

第2のテーブルは大局予測器と呼ばれる．これは分岐に至る実際の径路に基づいて分岐方向を予測するものである．この実行経路は11ビットのベクタによって特定される．これは，命令キャッシュから1回にフェッチされる命令の組(8命令)ごとに1ビットが割り当てられ，それまでにフェッチされた11回のフェッチの組を示す．このベクタは大局履歴ベクタ(global history vector)と呼ばれる．各ビットは，フェッチされる次の組が連続的なキャッシュのセクタからのものか否かを示す．大局履歴ベクタは，この情報を得て，実際の実行径路を知る．つまり，命令フェッチの方向が変更されるなら，その後に連続してフェッチされた命令の組は廃棄される．そして，同時に大局履歴ベクタが更新される．大局履歴ベクタと分岐命令の存在するアドレスのビットごとの排他的論理和は，16384エントリの大局履歴テーブルをインデックスし，もう1ビットの分岐予測情報を生成する．局所予測器と同様に，この1ビットの大局的予測情報は，分岐が成立すべきか否かを示す．

第3のテーブルは選択テーブルと呼ばれる．これは，与えられた分岐に対し，二つの予測機構のどちらが正しいかを判断する．そして，局所的と大局的予測の選択を行う．これは，大局履歴テーブルと同じ方式で16384エントリの選択テーブルがインデックスされ，1ビットの選択情報を出力する．このような分岐予測テーブルの組み合わせは，非常に正確な分岐予測を行うことができる．概念的には，大局履歴テーブルで次の命令の位置(キャッシュセクタ)を予測し，そこに条件分岐命令があれば，局所履歴テーブルで分岐方向が予測されるということであろう．実際には，これらの処理が同時に行われる．Power4の分岐予測方式は，通常の分岐予測と次にフェッチするキャッシュラインを予測して組み合わせる，Alphaの分岐予測方式によく似ている．

動的な分岐予測はソフトウェアによって上書きできる．これは，ソフトウェアがハードウェアよりも正確に予測できる場合に有用である．これは，条件分岐命令コードの中へあらかじめ決められた2ビットを設定することによって実現される．1ビットはソフトウェアによる上書きを指示し，もう1ビットは分岐の方向を指示する．もし，この2ビットが0なら，ハードウェアによる分岐予測が適用される．

● 命令フェッチ

命令は，命令フェッチアドレスレジスタ(IFAR：instruction-fetch address register)に基づき，命令キャッシュからフェッチされる．通常，IFARは，分岐予測機構によって決定されたアドレスを設定される．上述のように，分岐予測を誤るような場合は，分岐実行ユニットによって，フェッチされるべき正しいアドレスが設定される．さらに，命令の流れを変更する別の要因がある．これは，例外または外部からの割り込みによる．いずれにせよ，IFARに値が設定されたら，命令キャッシュがアクセスされ，1サイクルに最大8命令がフェッチされる．命令キャッシュの各ラインは32個(128バイト)のPowerPCの命令を保持する．各ラインは四つの同等なセクタに分割される．キャッシュミスは頻繁でないと仮定しているため，面積を節約するために，命令キャッシュは，各サイクルに1セクタをリード/ライトできる1ポートの設計になっている．命令キャッシュディレクトリ(IDIR：I-cache directory＝タグメモリ)は実効アドレスでインデクスされ，各エントリは42ビットの実アドレスを含む．つまり，仮想インデクス，物理タグキャッシュであることがわかる．

命令キャッシュミスが発生すると，命令はL2キャッシュから4回の32バイトの転送によってフィルされる．L2キャッシュはクリティカルセクタ(そのキャッシュラインを参照した特定のワードアドレスを含むセクタ)からリードされる．命令フェッチ機構は，この要求された命令をできるだけ早くパイプラインに投入する．さらに，残りのキャッシュラインは命令プリフェッチバッファに格納され，連続する命令フェッチに備える．

PowerPCアーキテクチャでは，実効(仮想)アドレスから実(論理)アドレスへの変換を行うために，TLB(Translation-Lookaside Buffer)とSLB(Segment-Lookaside Buffer)を定義している．これは2段階の構成を採用する．

アドレス変換には数クロックかかるので，1回変換が行われたら，実効アドレスと実アドレスの組を，128エントリの2ウェイセットアソシアティブ構成のERAT(Effective-to-Real Address Translation)テーブルと呼ばれる配列に格納する．Power4では，命令キャッシュとデータキャッシュに対応して二つのERATを実装する．それぞれは，IERAT，DERATと呼ばれる．これらのERATは実効アドレスでインデクス

される．これらは，俗に言うマイクロTLBである．

IERATやDERATは，各プロセッサに実装されている1024エントリの4ウェイセットアソシアティブ構成の共通TLBの内容をコピーしたものである．

命令パイプラインが命令受け付け可能になると，IFARの内容は，命令キャッシュ，IDIR，IERAT，分岐予測機構に転送される．次のサイクルで，命令キャッシュから命令が読み出され，命令キューに渡される．そして，デコード，クラック(crack)，グループ分け(group formation)が行われる．命令キューへの格納は命令キャッシュのヒットを待たずに行われる．同じサイクルで，IDIRから実アドレス，IERATから実効アドレスと実アドレスの組，予測された分岐方向の情報を受け取る．IERATはインデクスされたエントリが有効であることをチェックし，IDIRからの実アドレスが，IERATのエントリの内容と一致するかどうかチェックする．もし，IERATのエントリが無効なら，実効アドレスはTLBとSLBで変換される．この場合は命令フェッチは一時停止(ストール)する．IERATが有効で実アドレスが一致する場合，初めて命令キャッシュがヒットしたことが判明する．命令キャッシュがミスする場合でも，命令キューに格納された命令は(無効であるが)処理を継続され，パイプラインが止まることはない．

もし，キャッシュミスが発生すれば，次のように動作する．まず，命令プリフェッチバッファがチェックされ，要求された命令が存在すれば，あたかも命令キャッシュから読み出したかのように，その命令をパイプラインに入力する．もし，命令プリフェッチバッファに望みの命令が存在しなければ，L2キャッシュにリフィル要求(demand-fetch reload request)が発行される．L2キャッシュは，このリフィル要求を最優先で処理する．そしてL2キャッシュから来る命令は命令キャッシュに格納される．同時に，4エントリ(各エントリは32命令を格納)の命令プリフェッチバッファにも格納される．

● デコード，クラック(crack)，グループ分け(group formation)

命令はアウトオブオーダに実行されるので，「飛行中」のすべての命令に関し，プログラムでの順序を記憶しておかなければならない．その論理を最小化するために，命令はグループ化されている．そして，そのグループ単位で順序を管理される．つまり，プロセッサの状態は，グループ内の命令の切れ目ではなく，グループの切れ目で変化するのである．例外が発生した場合は，直前のグループの状態が保持される．

一つのグループは，IOPと呼ばれる内部形式の命令を最大5命令含む．デコードステージにおいて，命令は順序通りにグループに入れられる．もっとも古い命令がスロット0に置かれ，次に古い命令がスロット1に置かれる．同様にスロット2，スロット3が埋められる．スロット4が分岐命令用に予約されているが分岐命令が存在しない場合は，NOP命令がスロット4に入れられる．各サイクルで1グループのみをディスパッチすることができる．つまり，グループ内のすべての命令が同時にディスパッチされる．ここで，ディスパッチとは，命令のグループを発行キュー(リザベーションステーション)に移動することを意味する．グループはプログラムの順序でディスパッチされる．各IOPは発行キューから実行ユニットに対し，アウトオブオーダに発行される．その実行結果は，グループ内のすべての命令が完了したときに，コミット(リタイア)される．各サイクルでは一つのグループのみが完了できる．

正確な処理を行うために，ある種の命令は投機的に実行してはならない．それを保証するために，その種の命令は完了する最後の命令になるまで実行されない．この機構は，完了逐次化(completion serialization)と呼ばれる．実装を簡単にするために，その種の命令は一つだけで命令グループを形成する．完了逐次化の例として，ガードされた(順番を保証する)空間へのロード/ストア命令，プロセッサの状態を変更するマシン状態レジスタへの移動命令などのコンテキストの同期命令がある．

多くの場合，内部命令はアーキテクチャで見た命令と同じである．しかし，命令は一つ以上の内部命令に置き換えられる場合がある．高い動作周波数を達成するために，1命令でのレジスタアクセスは，リードは最大2個，ライトは最大1個に制限されている(3入力を必要とする積和命令などの浮動小数点命令の場合，性能を出すためにこの規則に従わない場合もある)．たとえば，load with update命令(1個のレジスタをリードした後にインデックスレジスタを増加させる)は，ロード命令と加算命令に分割される．同様に，複数のデータにアクセスする命令は，複数のロード命令に分割される．

グループ分けは二つのクラスに分類できる．load with update命令のように二つの命令に分割される場

表2 Power4の発行キュー

タイプ	エントリ数	キューの数
固定小数点/ロード/ストア	9	4
浮動小数点	5	4
分岐実行	12	1
CR論理	5	2

合は，その操作はクラックされた(割られた)と呼ばれる．もし，命令が3命令以上のIOPに分割されるなら，それらは粉砕された(millicoded)命令と呼ばれる．クラックされた命令は，他の命令と同様にグループの中に入れられる．ただし，二つのIOPは同じグループになるという制限がある．もし，二つの命令が，その時点のグループに入りきらないなら，そのグループは終了され，新しいグループから開始される．クラックされた命令に続く命令は，グループに余裕があれば，同じグループに入れることができる．粉砕された命令は，常に新しいグループから開始される．条件レジスタを操作する論理命令は発生頻度が小さいので，この機能のためには一つの実行ユニットしか用意されていない．また，グループの中ではスロット0とスロット1へ固定的に入れられる．

● グループのディスパッチと命令発行

命令のグループは，1回に一つのみ，発行キューにディスパッチされる．グループがディスパッチされると，そのグループに対応する制御情報がグループ完了テーブル(GCT：Group Completion Table)に格納される．GCTは最大20グループを格納できる．GCTのエントリにはグループ内にある最初の命令のアドレスが格納されている．命令が実行を終えるとき，その情報はGCTの対応するグループのエントリに登録される．終了情報は，グループがリタイアするまで(すべての命令の実行結果がコミットされるか，システムによってフラッシュされるまで)GCTに保持される．グループの各命令スロットは，浮動小数点演算ユニット，分岐実行ユニット，条件レジスタ(CR)実行ユニット，ロード/ストアユニットが独立に持つ発行キューに送られる．**表2**に発行キューの深さと個数をまとめた．浮動小数点演算キューと固定小数点(整数)，ロード/ストアユニットの共通な発行キューに対し，スロット0とスロット3に保持されている命令を転送できる．一方，スロット1と2は他の実行ユニットに転送される．CR実行ユニットへはスロット0と1から転送される．

表3 Power4のリネーム資源

資源のタイプ	論理サイズ(本数)	物理サイズ(本数)
GPR	32(36)	80
FPR	32	72
CR	8(9)	32
リンク/カウントレジスタ	2	16
FPSCR	1	20
XER	4領域	24

グループがディスパッチされるためには，そのグループで使用されるすべての資源が有効になっていなければならない．そうでなければ，必要な資源が空くまで待たされる．ディスパッチを行うためには，次の資源が有効であることが必要である．

1) GCTエントリ

各グループに一つのGCTエントリが割り当てられる．対応するグループがリタイアするときに解放される．

2) 発行キューのスロット

グループ内の各命令に対する発行キューのスロットが有効でなければならない．命令が実行ユニットに発行されたら解放される．

3) リネームレジスタ

各レジスタはグループ内の命令によってリネームされセットされる．対応するリネーミング資源が有効でなければならない(**表3**)．リネーミング資源は，同じ論理資源に書き込む次の命令がコミットしたときに解放される．

4) ロードリオーダキュー(LRQ：Load Reorder Queue)のエントリ

LRQのエントリはグループのすべてのロード命令に対して有効になっていなければならない．このエントリは，グループが完了するときに解放される．LRQは32エントリからなる．

5) ストアリオーダキュー(SRQ：Store Reorder Queue)のエントリ

SRQのエントリはグループのすべてのストア命令に対して有効になっていなければならない．このエントリは，グループが完了した後，ストアの結果がL2キャッシュに転送されたときに解放される．SRQは32エントリからなる．

上述のように，ある種の命令は完了逐次化が要求される．完全逐次化が必要なグループは，すべての先行するグループが完了するまで発行されない．さらに，

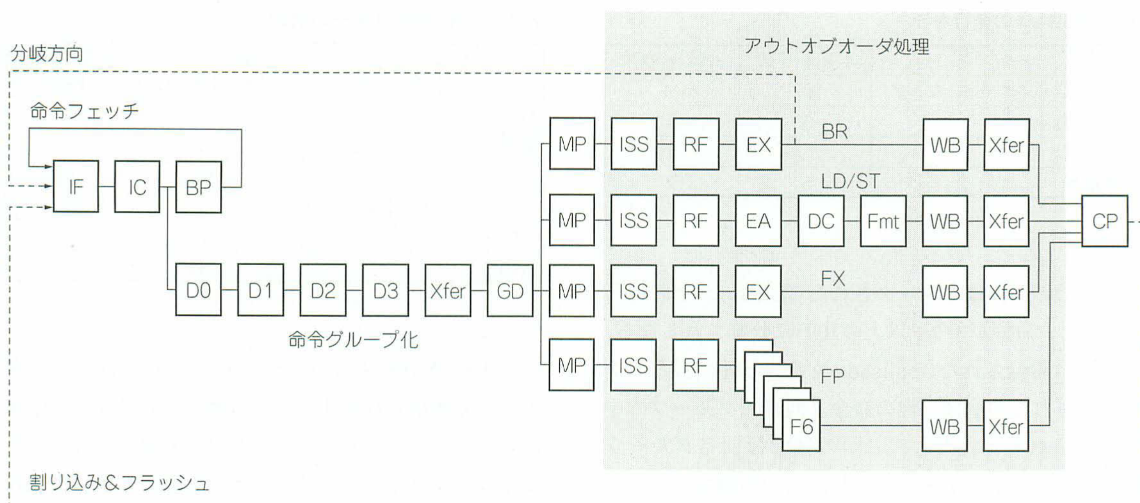


図22　Power4のパイプライン

リネームされていないレジスタをリードする命令は，そのレジスタへのライトが終了するまで実行してはいけない．実装を簡単にするために，リネームされていないレジスタにライトする命令の実行が終わるまで，そのレジスタをリードする命令のディスパッチが禁止される．リネームされていないレジスタへのライトはプログラムの順序で実行されることが保証されている．

命令がグループ単位で処理されていくため，グループ内のある命令が割り込みを受ける場合は，そのグループの命令と実行結果を無効化し，一つの命令からなるグループにして再ディスパッチする．

● 命令実行パイプライン

図22にPower4の命令実行パイプラインを示す．IF，IC，BPが，それぞれ，命令フェッチ，命令キャッシュ，分岐予測のサイクルに対応する．D0からGDサイクルは，命令デコードとグループ化したサイクルである．MPはマップを行うサイクルである．これは，依存性を調べて資源の割り当てを行い，グループを対応する発行キューにディスパッチすることである．ISSサイクル中に，IOP(内部命令)が各実行ユニットに発行される．RFはレジスタをリードするサイクル，EXは実行を行うサイクル，WBは実行結果をレジスタにライトバックするサイクルである．この時点で，命令の実行は終了するが，完了はしない．少なくとも，さらに2サイクルの間は完了できない．これはXferとCPサイクルがあるためである．ここで，先行するすべてのグループが完了したことと，同じグループのすべての命令が終了したかどうかを調べる．

命令キャッシュからフェッチされた命令は，D1サイクルにおいて命令キューの中で待ち合わせを行う．これは，命令は1サイクルに最大8命令をフェッチされるので，命令供給がグループ化よりも高速であるためである．同様に，命令はMPサイクルの前でも待ち合わせを行う．発行キューへグループをディスパッチするための資源の待ち合わせがあるためである．さらに，ISSサイクルの直前でも，命令は発行キューの中で待ち合わせを行う．また，CPサイクルの前でも命令の完了を待ち合わせる．

図示してはいないが，CR論理実行ユニット(条件レジスタに対する論理演算を受け持つ)は固定小数点(整数)パイプラインと同じである．固定小数点パイプラインは，**図22**中ではFXで示されている．

分岐実行ユニットのパイプラインはBPである．もし，分岐命令の予測が失敗した場合，少なくとも12サイクルのペナルティが生じる．

二つのロード/ストアユニットに対するパイプラインは同一で，**図22**中ではLD/STで示されている．レジスタファイルにアクセスした後は，ロード/ストア命令はEAサイクルで実効アドレスを生成する．ロード命令において，DERAT，データキャッシュディレクトリ，データキャッシュへのアクセスは，すべてDCサイクルで行われる．もし，DERATミスが発生するなら，ロード命令は拒絶される．つまり，発行キューの中に留まる．その間に，TLBへの要求が発行され，DERATがアドレス変換情報でリフィルされる．拒絶された命令は，最初に発行時よりも最小7サイク

ル後に再発行される．それでもDERATが変換情報を保持していない場合は，再び命令が拒絶される．この拒絶-再開処理は，DERATがリフィルされるまで続けられる．TLBミスが発生した場合，変換は投機的に行われる．しかし，TLBに変換情報が格納されるのは，アドレス変換が失敗した命令を含むグループの実行が完了する直前である．

ロードの場合，ディレクトリがL1データキャッシュが目的のキャッシュラインを含んでいることを示すならば，得られたデータの中から要求されたバイトを取り出し，Fmtサイクルで位置合わせを行い，指定されたレジスタにライトバックする．

ストア命令の場合は，ライトデータは，データキャッシュではなくて上述のSDQにライトされる．ストア命令を含むグループが完了したとき，ライトデータはデータキャッシュへライトされる．もしもL1データキャッシュの目的のキャッシュラインが有効なデータを含む場合は，L1データキャッシュが更新される．また，L1データキャッシュにラインが存在しないなら，そのラインがL2キャッシュからリフィルされることはない．どちらの場合も，データはL2キャッシュにライトされる．要するに，ライトアロケートなしのライトスルー構造である．

二つの固定小数点（整数）実行ユニットのパイプラインは，**図22**中では，FXで示されている．二つの固定小数命令が互いに依存性を持つ場合，連続したサイクルでは発行できない．この場合は，二つの発行サイクルの間に無効サイクルが最低1サイクル必要である．しかし，このサイクルで別の命令を発行することはできる．

二つの浮動小数点実行ユニットのパイプラインはFPで示されている．浮動小数点命令の実行には6サイクル必要（F1～F6）である．しかし，命令実行は完全にパイプライン化されている．つまり，各サイクルで連続的に2命令をFPに発行可能なのである．しかし，浮動小数点命令に依存性がある場合，命令実行の6サイクルの間に別の命令を発行することはできない．しかし，片方のパイプラインは独立して動作できる．

11 Pentium

● Uパイプ/Vパイプのスーパースカラ構造

Pentium（P5）のパイプラインは，i486と同様の5ステージから構成される．MMX Pentiumではフェッチステージが1段追加されて6ステージになる．イメージ的にはそのパイプラインが2本並列に動作するインオーダスーパースカラ構造である．二つの汎用整数パイプラインに加えて，パイプライン化されたFPU演算を同時に実行できる．

これら二つの整数パイプラインは，UパイプおよびVパイプと呼ばれる．Uパイプはすべての命令を実行できる．一方，Vパイプでは単純な命令のみを実行できる．同時発行可能な2命令をデコードしたとき，（プログラムの順番で）先行する命令はUパイプで，後続する命令はVパイプで実行される．イメージ的には，Uパイプが常に動作していて，後続命令が同時実行可能な場合のみVパイプも使用するといったところであろうか．

図23にPentiumのブロック図を示す．なお，五つのパイプラインステージの内訳は，次のようになっている（**図24**）．

1) PF　：プリフェッチ
2) F　　：フェッチ（MMX Pentiumのみ）
3) D1　：命令デコード
4) D2　：アドレス生成
5) EX　：実行（ALU演算とキャッシュアクセス）
6) WB　：ライトバック

● 各ステージについて

PFステージでは命令キャッシュまたはメモリから命令がプリフェッチ（先取り）される．Pentiumでは，従来のi486などとは異なり，キャッシュが命令キャッシュとデータキャッシュに分かれているので，プリフェッチがデータ参照と競合しない．PFステージでは，二つの独立なラインサイズ（16バイト×2）の組み合わせのプリフェッチバッファが分岐ターゲットバッファ（BTB）と結合されて動作する．条件分岐命令に行き当たるまでプリフェッチは逐次的に進む．条件分岐命令がプリフェッチされるとBTBで分岐予測が行われ，片方のプリフェッチバッファは分岐先のプリフェッチに使われる．これにより，分岐予測によるプリフェッチと同時に本来のプリフェッチを続行できる．MMX Pentiumでは四つの16バイトのプリフェッチバッファで最大四つの独立した命令の流れをプリフェッチ可能らしい．本当なのかと疑ってしまうが，これはユーザーズマニュアルからの受け売りである．

Fステージは MMX Pentiumのみに存在する．このステージでは命令の長さをデコードする．これは従来D1ステージで行われていた処理である．プリフィク

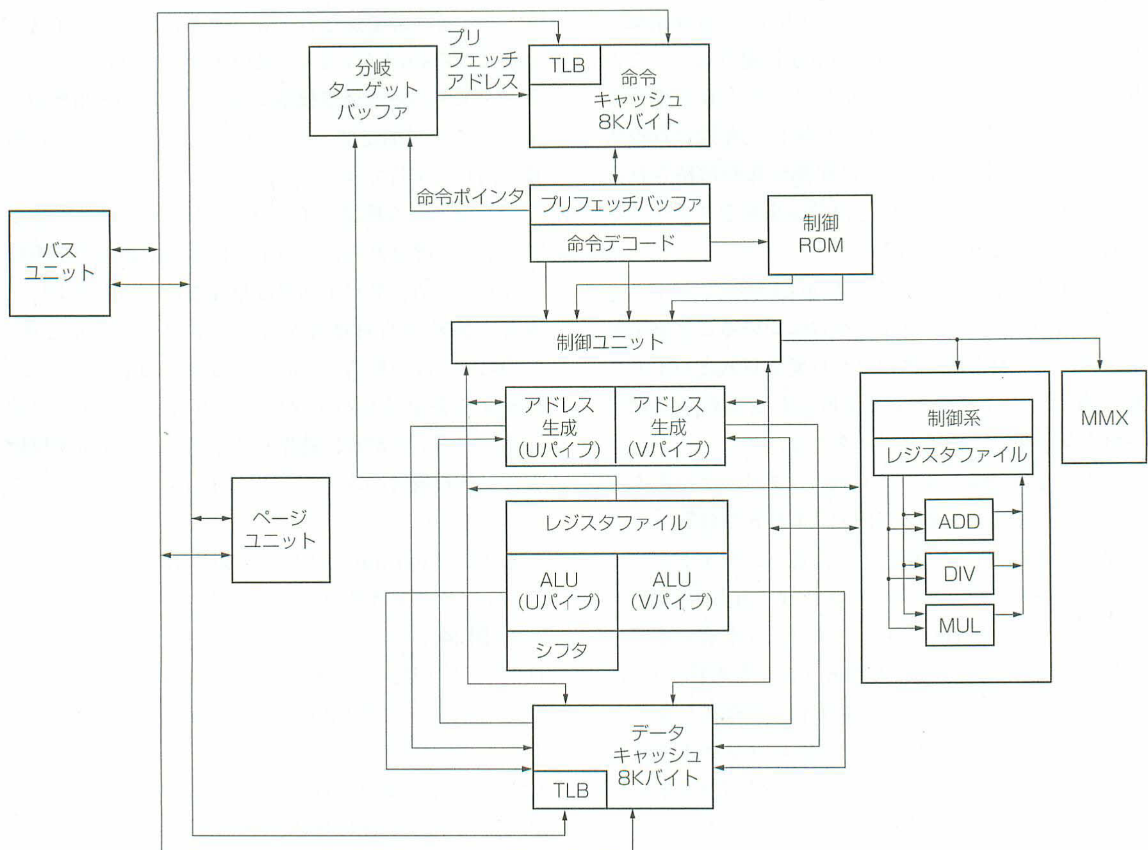


図23　Pentiumのブロック図

スのデコードもFステージで行われている．

さらに，MMX Pentiumでは，FステージとD1ステージの間に命令キュー(FIFO)が存在する．FIFOが空のときは，命令は遅延なしでD1ステージに渡される．FIFOは4命令分用意されており，各サイクルで2命令を格納可能である．FIFOからは2組の命令が引き出されてD1ステージに渡される．FIFOは通常命令で満たされているので，常に2命令を取り出すことが可能で，1サイクルで実行される平均命令数は2に限りなく近づく．FIFOがうまく機能している限りは，命令フェッチとFIFOからの命令の切り出しでストールは生じない．

D1ステージでは連続する2命令を同時にデコードし発行する．同時に発行できる命令の組は次のような関係にあるものである．

a) ハードワイヤード化された単純な命令
b) レジスタの依存性がない
c) ディスプレースメント付きとイミディエートの組でない
d) プリフィクスでない

なお，Fステージをもたない Pentiumではプリフィクスがある時だけD1ステージを繰り返す．また，プリフィクスは，他の命令と組になることはなく，Uパイプのみで実行される．すべてのプリフィクスが発行されると，ベースとなる命令(プリフィクスが付加されていた命令)は次の命令と同時発行が可能になる場合もある．

D2ステージではメモリオペランドのアドレスを計算する．また，レジスタオペランドをリードする．i486では，ディスプレースメントとイミディエートを同時に含む命令，または，ベースとインデクスを持つ命令はもう1回D2ステージが必要だったが，Pentiumでは不要になった．

EXステージはALU演算とデータキャッシュへのアクセスを行う．ALU演算とデータキャッシュアクセスの両方の処理が必要な場合，このステージではさらに1クロックが必要となる．EXステージでは分岐予測の正当性の検証も行う．ただし，Vパイプの条件分岐の検証はWBステージで行われる．また，マイクロコードで実行される複雑な命令はUパイプとVパ

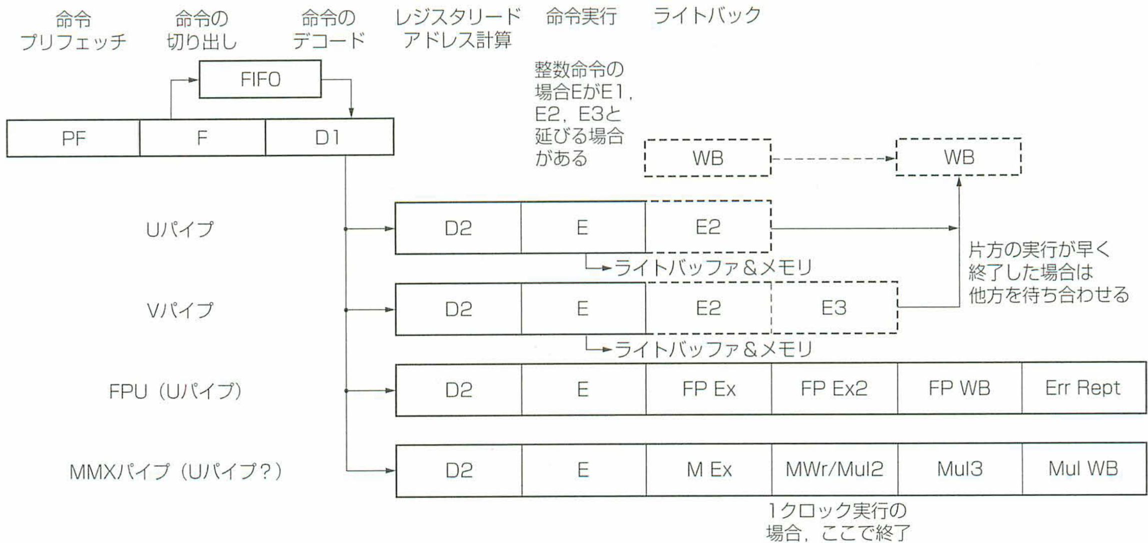


図24　Pentiumのパイプライン

イプの両方を使う．

WBではプロセッサの状態を更新して実行を完了する．

UパイプとVパイプで実行される命令は，同時にD1，D2ステージに入り，同時に抜けていく．当然，EXステージにも同時に入る．もし，片方のパイプがストールすれば他方のパイプもストールする．両方のパイプの命令がWBステージに達するまで，新たな命令はEXステージに入って来られない．こうして，インオーダ完了を実現している．

12 Pentium II

● 複雑なスーパーパイプライン構造

Pentium II（P6）のパイプラインは複雑である．動作周波数を上げるためにスーパーパイプライン構造を採り，IPCを上げるためにアウトオブオーダのスーパースカラ構造を採っている．14ステージのパイプラインは次の三つのセクションに分割できる．そして，これらのセクションは独立して動作する（**図25**）．

a）インオーダな前処理（8ステージ）

b）アウトオブオーダ実行（3ステージ）

c）インオーダなリタイア（3ステージ）

パイプラインのステージ数は，機能分割の方法により12ステージ，または10ステージという説もある．ここでは，Pentium IIが発表された当時の一般的な解説記事にしたがおう．Intelの公式資料ではステージ数は明記されていなかったと記憶している．マニュアルにある図のステージ数を数えると13ステージのようにも思えるが，実行ステージを抜いて12ステージという解釈が有力であった．なお，Pentium4の発表に当たり，Pentium IIのパイプラインステージ数は公式に10ということになった．

Pentiumは（かなり制限のある）2命令同時発行のMPUだったが，Pentium IIでは3命令同時発行になった．単にパイプラインの本数を増やしただけでなく，Pentium IIはx86命令をμOPというRISC風の固定長命令に変換し，効率的にパイプラインを処理した．x86命令の欠点（?）として，エンコード（命令コードのビット並び）に規則性がないこと，レジスタ-メモリ間演算，可変長命令などが挙げられるが，従来はこれらの特徴が効率的なスーパースカラ処理の妨げとなっていた．μOPを導入することで命令のデコードが容易になり，RISC並みのパイプライン効率を得ることができる．

図26にPentium IIの機能ブロックを示す．この図を基に命令処理の過程を説明しよう．

● x86命令の変換

x86命令からμOPへの変換は，パイプラインの最初の8ステージで行われる．まず，分岐ターゲットバッファ（BTB）が指し示す位置の64バイト（キャッシュ2ライン分）のコードを命令キャッシュから読み込む．その中で，最初にあるx86命令の先頭から16バイトのコードを取り出して，並列動作する三つのデコーダ

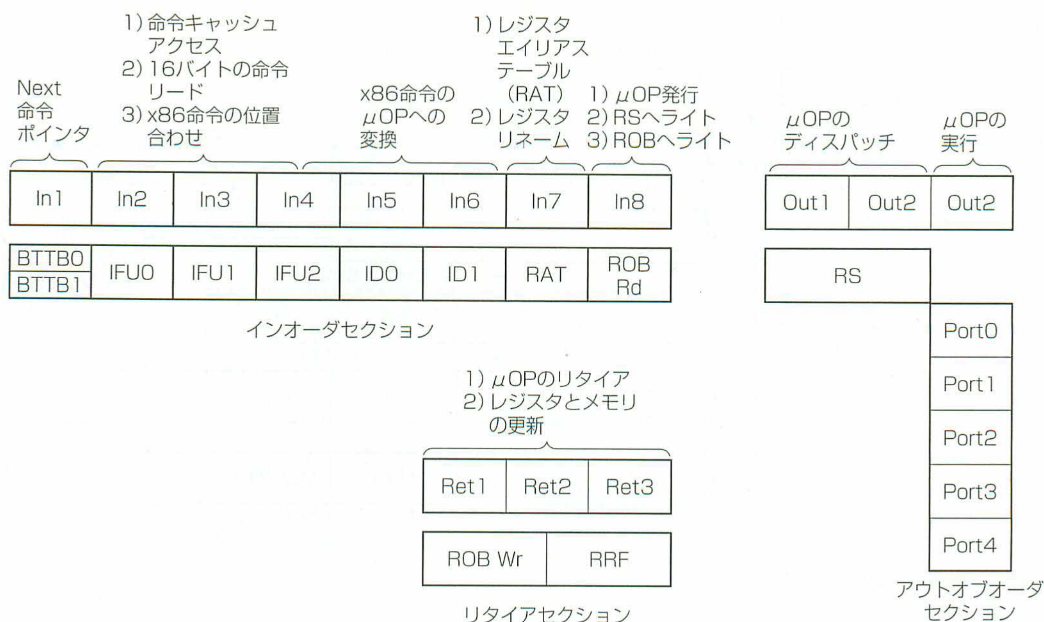


図25　Pentium Ⅱのパイプライン

に渡す．x86アーキテクチャは可変長命令を採用し，プリフィクスを付加することで(理論上)無限長の命令を生成することができる．

Pentium Ⅱは，1命令の長さを平均5バイトと仮定しているのであろう．もっとも，16バイトのコードのうち，この時点で命令の切れ目は不明なので，三つの命令デコーダは16バイトすべてを受け取ると推測される．Pentium系のMPUでは，命令が16バイト境界にまたがる場合は命令の実行効率が落ちるといわれているが，それはここに原因があると思われる．

さて，Pentium Ⅱの命令キャッシュは1ラインが32バイトなので，その中の任意の位置から始まる16バイトのコードを得るために，二つのラインが同時に読み込まれる．そして，これら3種の命令デコーダがx86命令をRISC命令によく似たμOPに変換する．三つのデコーダのうち，二つが単純デコーダ，残りが複雑デコーダである．単純デコーダは一つのx86命令を一つのμOPに変換し，複雑デコーダは一つのx86命令を一つから四つのμOPに変換する．とくに複雑な命令は複雑デコーダでもデコードできず，そこを通過してマイクロコード命令シーケンサ(MIS)に渡される．MISは必要なだけのμOPを生成する．

たまたま複雑な命令が単純デコーダに割り当てられる場合は，そこから複雑デコーダまたはMISに渡される．ここでのデコードの遅れはリザベーションステーションで吸収されるので，命令の実行には影響しない．

単純な命令と複雑な命令のデコーダへの割り当てが完璧な場合は，1サイクルごとに六つのμOPを生成する．平均すると1サイクルごとに三つのμOPが生成されている．これを根拠に，IntelはPentium Ⅱを「3ウェイスーパースカラ」と呼んでいる．

● レジスタリネーム

μOPに変換されたx86命令は，パイプラインの第7ステージでレジスタエイリアステーブル〔Register Alias Table：レジスタ読み替え表(RAT)〕に送られてレジスタリネームが行われる．ここで偽の依存性(WAWハザードなど)を解消する．

x86アーキテクチャは汎用レジスタ(論理レジスタ)が8本しかないので，レジスタの依存関係が生じやすい．それを軽減させるため，Pentium Ⅱでは40本の物理レジスタをもつ．つまり，Pentium Ⅱは内部的に40本の汎用レジスタをもっていることになる．

レジスタリネームでは，真の依存性(RAWハザードなど)は解消できない．しかし，Pentium Ⅱではレジスタのフォワーディングを行うので，そのペナルティを軽減できる．

● アウトオブオーダ実行

レジスタリネームが完了すると，プログラムの順序通り，μOPはリオーダバッファ(ROB)に送られると

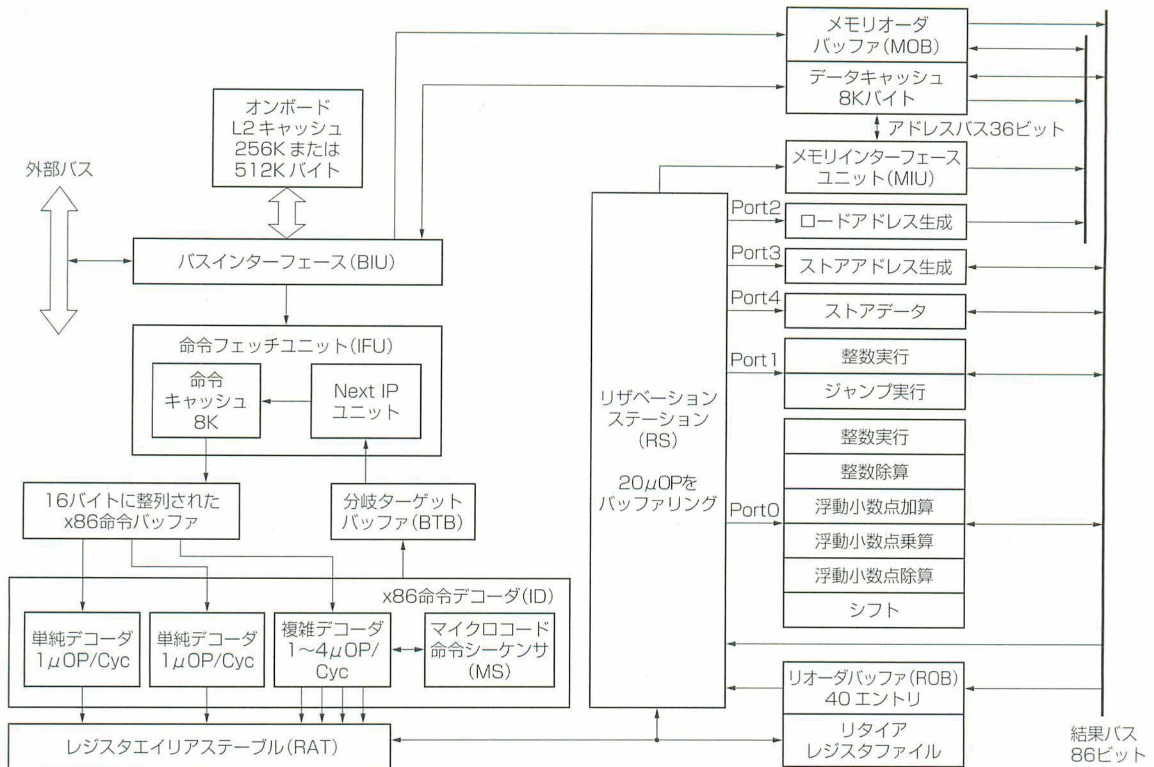


図26 PentiumⅡのブロック図

同時にリザベーションステーションにキューイング(待ち行列に入れる)される．これは，デコーダと実行ステージの中間に位置する．リザベーションステーションは最大20個のμOPを蓄えることができ，11個の実行ユニットに対して，1サイクルで最大五つのμOPを発行できる(入力ポートが五つあるため)．もっとも，典型的なx86の命令列では1サイクルに発行できるμOPはたかだか3命令といわれている．

リザベーションステーションは，ソースオペランドが使用可能になったか，実行ユニットが空いたか，依存性が解消できたかなどを調べて，用意ができたμOPをアウトオブオーダに発行する．アウトオブオーダにコンプリートするμOPの結果は，一時的なバッファ(ROBやMOB)に格納され，ROBの状態を参照しながらプログラムの順序に従ってレジスタやメモリに書き込まれる．

ROBは40エントリからなる254ビット幅のバッファである．254ビットの内訳は，二つのオペランド，実行結果，多くの状態ビットである．ROBには整数と浮動小数点のμOPの両方が格納される．ROBやMOBから取り出す処理はパイプラインのリタイアのステージで行われる．

● リタイア

ROBは実行状態と各μOPの結果を保持する．μOPは，先行するμOPがすべてコンプリートしたことがわかって初めてリタイアし，結果をレジスタやメモリに書き込む．この動作を「コミット」ともいう．PentiumⅡでは1サイクルに最大三つのμOPをリタイアできる．これは，デコーダが1サイクルに発行できる平均的なμOPの個数(3命令)と釣り合いが取れている．

オペランドのフォワーディングのために，それぞれの実行ユニットの結果はすべてリザベーションステーションに戻される．実行ユニットの結果はROBにも戻されて，リタイアの準備ができたか否かを決定する．

レジスタに対する結果はROBに書き込まれるが，メモリに対する結果はメモリオーダバッファ(MOB)に書き込まれ，対応するμOPがリタイアするまで一時的に格納される．メモリライトを生じるμOPがリタイアして初めてMOBはメモリにデータを書き込む．

● 分岐予測は必須

PentiumⅡはパイプラインのステージ数が多いの

で，分岐予測は必須である．分岐予測を誤った場合のペナルティは4～15サイクルである．これはかなりの性能低下になるので，高度な分岐予測が要求される．

Pentium(P5)と同様，Pentium IIは分岐ターゲットバッファ(BTB)を採用する．予測方式は分岐履歴ビットによる．一つの分岐先アドレスに対して，過去4回分の履歴を記録しておき，それに従って予測する．これは基本的にPentiumと同じで，4回のループならほぼ100％の分岐予測が可能だという．BTBにヒットしない分岐命令はオフセットの正負などから静的な分岐予測を行う．Intelの主張によると，分岐予測の正確さは，Pentiumが80％だったのに対してPentium IIは90％だという．逆にいえば，分岐予測を誤る確率は20％から10％へと半分になったということである(数字のマジック？)．

Pentium IIで採用している分岐予測の方式は2レベル適応履歴アルゴリズムというものだが，詳細は明らかにされていない．ただし，命令キャッシュのラインごとに四つの分岐先アドレスをBTBで予測することが公表されている．

ちなみに，先に示したNetNewsで報告されたDhrystoneベンチマークの計測結果では，BTBのミス率は33％だという．分岐予測自体の正確さは98％というから(BTB)にヒットする限り，分岐予測はうまく働いているようである．ただし，Dhrystoneのような単純なプログラムでBTBに67％しかヒットしないというのは納得がいかない．BTBは512エントリなので，1回目のループですべての分岐命令はBTB内に取り込まれ，後は98％の確率で分岐予測が成功するというシナリオを誰でも思い浮かべるはずである．多分，BTBの処理アルゴリズムに欠陥があるのかもしれない．

● 投機実行

Pentium IIも，分岐予測を有効に活用するために投機実行を行う．Pentium IIでは，分岐予測が失敗した場合の回復処理は，投機的に実行された命令に対するROBのエントリを無効化することで実現している．Pentium IIでは他の多くのMPUと同様に，一つ以上の分岐の方向を予測し実行していくという，多重レベルの投機実行を許している．ただし，ROBが一気に無効化されるため，分岐予測失敗時のペナルティは非常に大きい．

Pentium IIではサブルーチンに対するCALL/RETの組を高速に実行する機構をもっている．サブルーチンはプログラムのさまざまな場所から呼ばれるため，RET命令の分岐先を予測するのは難しい．Pentium IIではリターンスタックと呼ばれる機構でRET命令の分岐先を予測する．これはCALL命令のデコード時に戻りアドレスを格納するスタックである．RET命令をデコードするとリターンスタックにあるアドレスから分岐先を取り出して，そのアドレスを予測したアドレスとして命令フェッチする．物理的なスタックの内容は他の命令で変更されるおそれがあるので，リターンスタックのアドレスはあくまでも予測値でしかないことに留意すること．なお，これはスタックキャッシュとして昔から知られている手法でもある．

すでに説明したR10000もそうであるが，命令のデコードを容易にするために，命令をプリデコードした結果を命令キャッシュに格納するという手法が今後の流行になるかもしれない．そう言えばTransmetaのCrusoeも，x86命令をVLIW命令に変換してDRAM上にマッピングされた命令キャッシュ(?)に格納している．AMDはK6ですでにプリデコードしてある命令長情報をキャッシュに入れる構造になっていた．

● Pentium IIIはPentium IIと同じ

Pentium IIの後継にはPentium IIIがある．どちらもP6アーキテクチャを実装するので，パイプラインの構造などには違いはない．Pentium IIIはPentium IIの動作周波数の向上のほか，SSEというSIMD命令の追加とPSN(Processor Serial Number)の採用を特徴とする．PSNはプライバシ保護の観点から非難を浴び，その後継のPentium4では削除された．

13 Pentium4

● ハイパーパイプライン

Intelの開発した**Pentium4**(コード名Willamette)は，Pentium IIIに続く製品として2000年に登場した．2 GHz以上の動作周波数を目指し，従来の倍のパイプラインステージを採用したため，同一動作周波数ではPentium IIIよりも性能が劣る．このような状況は，アーキテクチャの変更時には多々あることであり，避けて通れない道でもある．しかし，何だかんだいいながらも，現在のIA-32プロセッサの主流はPentium4である．

Pentium4のマイクロアーキテクチャは，「NetBurst」と呼ばれる．ここでは，NetBurstのパイプラインの概要について述べる．**図27**にPentium4のブロック図を示す．Pentium4のパイプラインは次の三つの部分から構成されている．これは，Pentium IIのパ

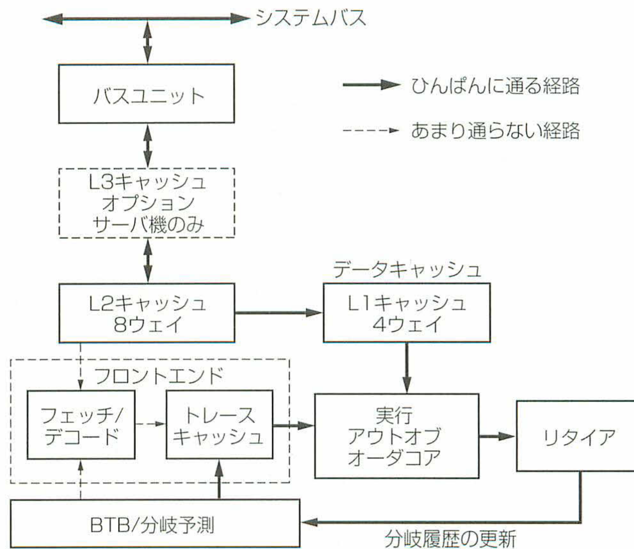


図27 Penitum4のブロック図

イプラインと同じである．

▶**インオーダな発行を行うフロントエンド**

▶**アウトオブオーダなスーパースカラ実行コア**

▶**インオーダなリタイアユニット**

フロントエンドはプログラム順の命令をアウトオブオーダな実行コアに供給する．つまり，IA-32命令をフェッチし，デコードし，マイクロ操作(μOP)に変換する．フロントエンドの主要な仕事は，μOPの連続的な流れを本来のプログラムの実行順序で実行コアに供給することである．

実行コアは1クロックに複数のμOPを発行し，そのμOPの入力の準備ができ，実行に必要なハードウェア資源が利用可能なものから，μOPの順序を入れ替えて実行する．

リタイア部はμOPの実行結果が本来のプログラムの順序に従って処理されることを保証し，必要なアーキテクチャ上の状態を更新する．

● パイプラインステージ数

Pentium4の発表にともない，IntelからNetBurstのパイプラインが公表された(**図28**)．Intelの公式見解では，Pentiumのパイプラインは5ステージ，Pentium IIのパイプラインは10ステージ，NetBurstは20ステージということになったようだ．20というステージ数は従来のスーパーパイプラインを超えるという意味で「ハイパーパイプライン」と呼ばれている．各ステージの具体的な動作に関しては公式な説明がない．ステージの名称から推測するしかないが，けっこう複雑なことをやっているような気がする．

パイプラインのステージ数が増えた理由は，動作周波数を向上させるためである．Pentiumが233MHz動作，Pentium IIが1GHz程度の動作であるのに対し，NetBurstでは1.4GHz動作を初めとして3GHz以上の動作を達成できるといわれている．

ただし，Pentium4は分岐予測機能を強化して深いパイプラインステージ数に起因する分岐命令の予測ミ

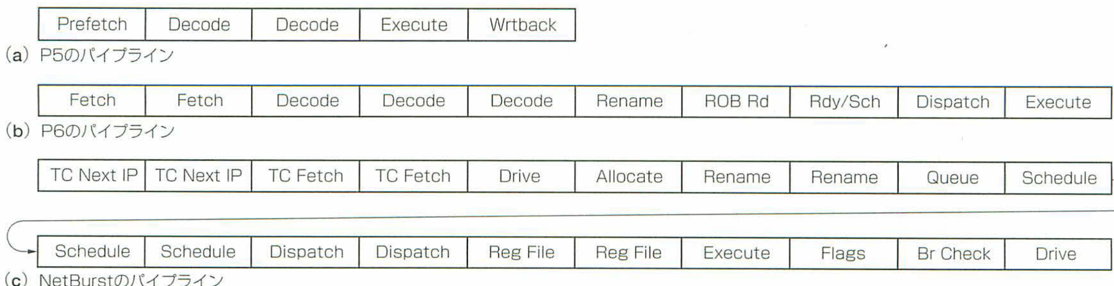


図28 Pentium4のパイプライン

ス時のペナルティを軽減したにもかかわらず，同一動作周波数のPentium Ⅲと比較すると性能が劣るというのは周知の事実である．パイプラインのステージ数の増加が性能に与える影響はかなり大きいことがわかる．それにもかかわらずステージ数の増加に至ったのは，Pentium Ⅲのクリティカルパス(回路のスピードネックとなる箇所)潰しが限界にきていることを意味する．つまり，Pentium Ⅲ(およびPentium Ⅱ)アーキテクチャでは2GHz以上の動作周波数が達成できないということである．この意味からPentium4(NetBurst)は動作周波数が2GHzを超えて初めてその存在価値がでてくるわけである．

ところで，NetBurstのIPCが低いということは，パイプラインがスカスカであることを意味する．これはHyperThreadingを実現するためという意見もあるが，真偽は不明である．

● 二つの部分からなるフロントエンド

フロントエンドは，二つの部分からなる．それは，

- フェッチ/デコードユニット
- 実行トレースキャッシュ

である．また，フロントエンドは次の基本機能を実行する．

- 実行すると予想されるIA-32命令をプリフェッチする
- プリフェッチされていない命令をフェッチする
- 命令をデコードしμOPに変換する
- 複雑な命令と特殊用途のコードに対しマイクロコードを生成する
- デコード済みの命令を実行トレースキャッシュから供給する
- 高度なアルゴリズムを用いて分岐予測を行う

さらにフロントエンドは，高速なパイプライン処理に関する一般的な問題のいくつかに注目している．たとえば，次の二つの問題に起因する遅延がある．

- 分岐先からフェッチする命令のデコード時間
- キャッシュラインの中間に位置する分岐や分岐先に起因するデコードの負荷

実行トレースキャッシュは，デコードしたIA-32命令を格納することで，これら二つの問題を解決できるように設計されている．命令は変換エンジンによってフェッチされデコードされる．変換エンジンはデコードされた命令を用いて，トレースと呼ばれる一塊のμOPに変換し，実行トレースキャッシュに格納する．実行トレースキャッシュは，これらのμOPをプログラムの実行順序にしたがって格納する．そこでは，コード中に出現する条件分岐の結果(分岐先または分岐元の命令)は予測されて同一のトレースキャッシュのラインに格納される．これにより，分岐によって実行されない命令を格納しないため，キャッシュ容量の効率的な利用が可能になる．あるいは，実行トレースキャッシュは分岐命令をある程度削減しているので，分岐によるペナルティをあらかじめ低減する意味もあると思われる．

実行トレースキャッシュは，実行コアに1クロックに最大三つのμOPを供給できる．この実行トレースキャッシュと変換エンジンは連動する分岐予測ハードウェアと連動している．分岐先はそのリニアアドレスに基づいて予測され，できるだけ早くフェッチされる．分岐先は，もし実行トレースキャッシュにキャッシュされているなら，そこからフェッチされる．もしキャッシュされてない場合は，外のメモリ階層(L2キャッシュなど)からフェッチされる．変換エンジンの分岐予測は実行されると予想される経路にしたがってトレースを形成する．

● 実行トレースキャッシュの構造

さて，Intelが出願している米国特許6,014,742にしたがって，実行トレースキャッシュの構造を推測してみる．トレースキャッシュは**図29**のような構成をしている．ある程度の数の命令(μOP)を実行順に(予測して)並び替えたものがトレースである．

トレースを形成するとき，分岐予測にしたがって動的に実行される命令の流れを追っていくが，それは無限に継続するのではなく，ある程度進んだ時点で中断する．単純には分岐から分岐までを一つのトレースとして形成すればいいのだが，前記の公開特許ではトレース内に条件分岐命令が含まれることを想定している．

トレースを実際にどの時点で中断するのかはよくわからない．実行トレースキャッシュはこのトレースをキャッシュしたものであり，各トレースは3μOP(公開特許では6μOP)からなるトレースラインから構成されている．実行コアには現在のトレースの中からトレースラインの内容を順次実行コアに送っている．

実行トレースキャッシュにはTBTB(Trace Branch Target Buffer)と呼ばれる独立した分岐予測機構がつながり，実行コアに与えるトレースラインごとに分岐予測を行い，トレースラインの供給を継続するか中断するかを決定する．この分岐予測は，トレース形成時の分岐予測とは独立していて，二つの予測が一致する

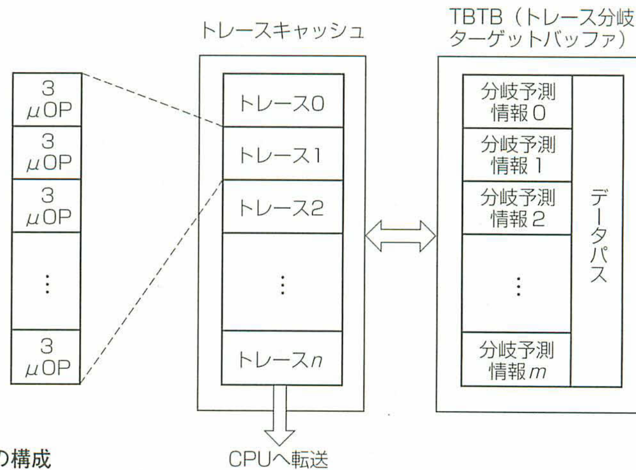


図29　Pentium4のトレースキャッシュの構成

ときのみトレースラインの供給を継続する．二つの予測が異なる場合は，キャッシュされているトレースに新たな分岐先があるか否かを探し，トレースキャッシュ内にあれば(要はトレースキャッシュにヒットすれば)，該当するトレースラインを遅延なしで実行コアに供給する．

実行コアから見れば，分岐予測が非常に正確に行われており，正しい経路の命令が供給されているように見える．新たな分岐先が実行トレースキャッシュになければ(トレースキャッシュミス)，トレース単位で不要なものと入れ替えが行われる．命令がループになっている場合は同じトレースに何度もヒットすると考えられる．

だいたいこのような感じであるが，以上は公開特許からの筆者の想像なので，現実の実装と異なっていてもご容赦を願いたい．

1トレースラインに3 μOPが含まれるということは，12,000命令を格納するというトレースキャッシュは4,000ラインから構成されることになる．このトレースラインをいくつか寄せ集めたものがトレースである．なお，トレースキャッシュの容量は，μOPが12,000命令ということであるから，x86命令に変換すれば16Kバイト相当といわれている．

ところで，実行トレースキャッシュの発想は，TransmetaのVLIWプロセッサであるCrusoeのトレースキャッシュとよく似ている．命令を実行するコアはスーパースカラとVLIWという違いがあるものの，x86命令を実行コアが都合のいい別の形態にプリデコードしてキャッシュするというものである．プリデコード時に分岐命令の挙動を予測し，プログラムの順序ではなく，実行する順序に並び替えてキャッシュするところもそっくりである．この並び替え操作をPentium4はハードウェアで実現するが，CrusoeはCMS(Code Morphing Software)で実現する．

● アウトオブオーダ実行コア

命令をアウトオブオーダに実行するコアの機能は並列性を可能にする主要な要素である．この機能は，一つのμOPがデータや関連する資源を待つ間に待ち合わせが必要なら，プログラムの順序では後に現われる他のμOPを先行して処理させるように，命令の並び替えを可能にする．

プロセッサはμOPの流れをスムーズにするためのいくつかのバッファを備えている．これは，プロセッサのパイプラインの1ヵ所が遅延しても，並行に実行している他の操作や(コアでの効果)，先立ってバッファにキューイングされているμOPの実行(フロントエンドでの効果)によって，その遅延が埋め合わされることを意味している．

実行コアは並行実行が可能なように設計されている．四つの発行ポートを通じて，1クロックに最大六つのμOPをディスパッチ可能である．発行ポートを**図30**に示す．1クロックに6命令のμOPの発行することは，トレースキャッシュやリタイアユニットの処理能力を超えていることに注意したい．これにより，ピークの処理能力を3 μOPより大きくし，異なる発行ポートへのμOPの発行に柔軟性を持たせることでより高い発行の割合を実現している．

ほとんどの実行ユニットは各サイクルで新しいμOPの実行を開始できる．このため，同時に複数の命令がそれぞれのパイプラインで処理状態になる．算術

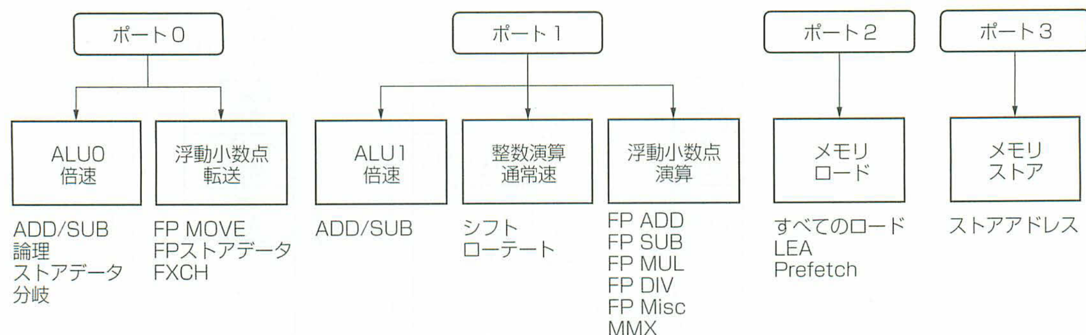


図30 Pentium4の実行ユニットとアウトオブオーダコアのポート

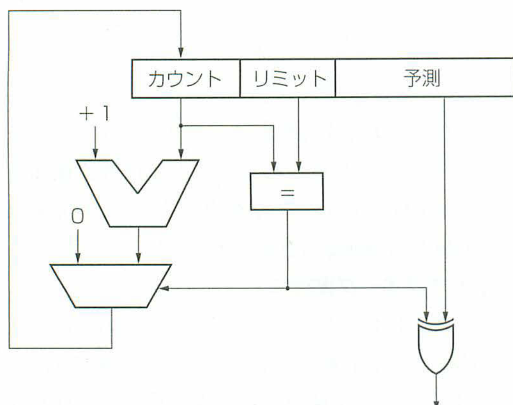


図31 Pentium Mのループ検出の論理

論理演算ユニット(ALU0/ALU1)を用いる多くの命令は1クロックに2命令を開始できる(倍速で動作する). また, 浮動小数点演算命令の多くは2クロックに1命令の割合で開始できる. μOPは, その入力データの用意ができて資源が利用可能になれば, 直ちにアウトオブオーダに実行を開始できる.

● リタイア

リタイア部は, 実行されたμOPの結果を実行コアから受け取り, その結果を本来のプログラム順序にしたがってアーキテクチャ上の状態を正常に更新する. IA-32命令の結果は, リタイアする前に, 意味的に正しい実行のために, 本来のプログラムの順序でコミットされなければならない. 例外は命令がリタイアするときに発生する. 例外は投機的には発生せず, 正しい順序で発生し, プロセッサは例外処理後に正しい位置から再開される. 一つのμOPがコンプリートし, 結果をデスティネーションにライトするとリタイアである. 1クロックに最大三つのμOPをリタイアできる.

リオーダバッファ(ROB)は, コンプリートしたμOPを格納し, アーキテクチャ上の状態をインオーダに更新し, 例外の順序を管理するユニットである. リタイア部は, 分岐を追跡し, 分岐先の情報を分岐ターゲットバッファ(BTB)に送り, 分岐の履歴を更新する.

14 Pentium M

● x86最新アーキテクチャ

Pentium M(Banias)に関する情報はほとんど公開されていないが, 2003年5月21日発行の*Intel Technology Journal*のVol.7 Issue 2の3番目の記事にPentium Mのマイクロアーキテクチャの解説がある. これを読んでも, マイクロアーキテクチャに不明な点は多い(パイプラインなど)が, Advanced Branch Prediction(進んだ分岐予測), μOPフュージョン, Dedicated Stack Engine(専用スタックエンジン)についての解説がある. これらについて簡単に解説しておく.

● Advanced Branch Prediction

Pentium Mのマイクロアーキテクチャは Pentium Ⅲに基づいているというのが定説だが, 分岐予測に関してはPentium4の技術を採用しているらしい. 特殊なプログラムの流れを追うため, IP(Instruction Pointer = Program Counter)に基づいて分岐先の成立/不成立を予測する通常の分岐予測機構のほかに, ループ検出器(Loop Detector)と間接分岐予測器(Indirect Branch Predictor)を備える.

ループ検出器はループ動作を検出し, その分岐先を予測する. ループは一定の回数同じ方向に分岐し, 1回だけ逆方向に分岐する. このため, ループ回数が判明していればループ操作を完全に予測できる(**図31**). ループが検出されると分岐予測機構の中に1組のカウンタを割り当てて回数を計数していくが, ループ回数をどのように決定するのかは明らかにされていない.

間接分岐予測器は，プログラムの流れによってデータ依存のある間接分岐を解消する．間接分岐は，オブジェクト指向コード(C++やJava)で多用されるが，その分岐予測が誤れば分岐予測性能の低下につながる．

ほとんどの間接分岐は，実行時には同一の分岐先に分岐する傾向がある．しかし，JavaのバイトコードのインタプリタやC++のCASE文はデータに依存して複数の分岐先をもつ．

間接分岐器は，IPで参照する分岐ターゲットキャッシュと，大域履歴(Global History)でインデックスする分岐ターゲットキャッシュをもつ．IPでの予測が成功した場合はその分岐先を使用し，IPでの予測が外れた場合は大域履歴の分岐先を使用する(**図31**)．

大域履歴は，IPに基づいた予測の付随物であり，IPでの予測が外れると大域履歴に登録する．つまり，過去何回かの間接分岐の分岐先を記憶しておき，もっとも確率の高い分岐先を選択する．

Pentium Mの分岐予測は前の世代の設計(Pentium IIアーキテクチャ)よりも，予測を外す確率が20%低下しており，実際の性能は7%向上しているという．この向上率の約30%はループ検出器と間接分岐予測器の組み合わせが寄与しているだろう．

● μOPフュージョン

x86プロセッサでは，IA-32の命令(マクロ命令)をμOPと呼ばれるRISC命令に変換して，RISCエンジンで実行することがなかば常識である．しかし，一つのマクロ命令は複数のμOPに分解されるので，リネームやリタイアのバンド幅やリオーダバッファやリザベーションステーションの容量といったハードウェア資源の不足を招く．そしてこれが性能低下に結び付く．それを解消するための手段がμOPフュージョンである．

基本アイデアは，複雑な操作(3個以上のオペランドが必要な操作)を行うマクロ命令も一つのμOPPとしてデコードして割り当て，リネーム，リオーダバッファやリザベーションステーションへの登録を行うということである．これがフュージョン(融合)ということらしい．融合というよりは，分解しないといったほうが正確である．

従来のμOPは2個のオペランドしかもてなかったので，3個以上のオペランドが必要なマクロ命令は2個以上のμOPに分解していた．融合されたμOPをサポートするために，Pentium Mでは，リザベーションステーションの各エントリは最大3個のソースオペランドを収容できるようになった．また，マクロ命令とμOPの対応が1対1になるので，命令デコーダも，複雑デコーダだけでなく，単純デコーダだけですべての命令デコードが可能になるという．

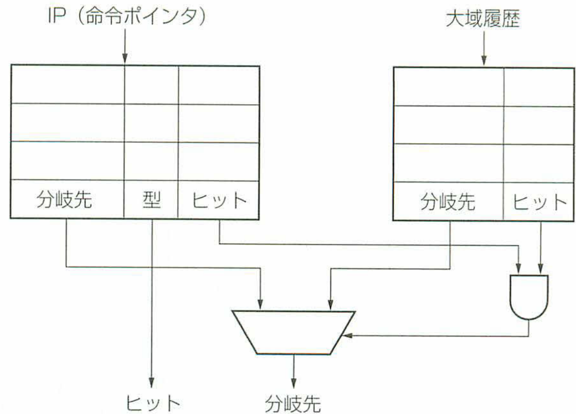


図32 Pentium Mの間接分岐予測器の論理

リザベーションステーションに格納されたμOP命令は，実行ユニットへのディスパッチ時に本来の2オペランドの複数のμOPに変換される．そして，分解された従来と互換性のあるμOPが実行ユニットでアウトオブオーダに実行される．一つの融合されたμOPを構成する複数のμOP(従来形式)がすべて完了すると，その融合されたμOPがリタイアする(**図32**)．

上述の論文では，μOPフュージョンの例として，ストア操作とリードモディファイ(load-and-op，リードした値と演算する)操作が挙げられている．これらのマクロ命令は，ディスパッチ時に2個のμOP(従来形式)に変換される．ストア操作は「ストアアドレス操作」と「ストアデータ操作」に分解される．リードモディファイ操作は「ロード操作」と「演算操作」に分解される．なお，マクロ命令が2個のμOPに分解される場合は稀であるとされている．

融合されたストア命令を形成する2個のμOPは並列に発行できる．メモリへの実際のライトはストア命令がリタイアされたときに行われるので，それまでにストアデータバッファに対してアドレスとデータが供給されていればいい．ストアアドレス操作はアドレス生成ユニットへディスパッチされ，そのソースオペランド(ベースやインデックスレジスタ)が用意されたときに実行される．ストアデータ操作はストアデータバッファユニットにディスパッチされ，そのソースオペランド(ストアするデータ)が用意されたときに実行さ

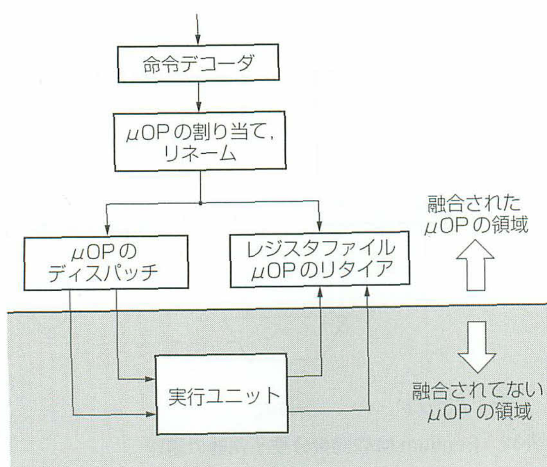


図33　Pentium MのμOPフュージョンの領域

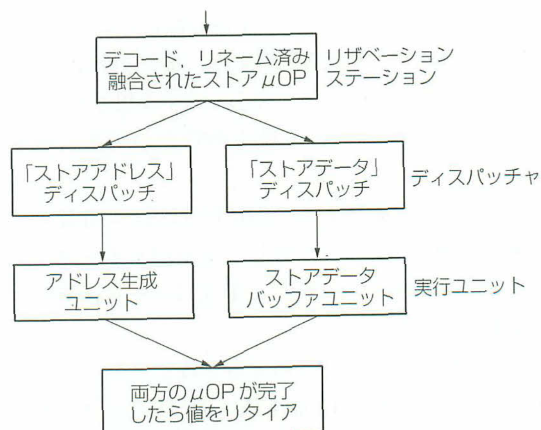


図34　Pentium Mの融合されたストアの流れ

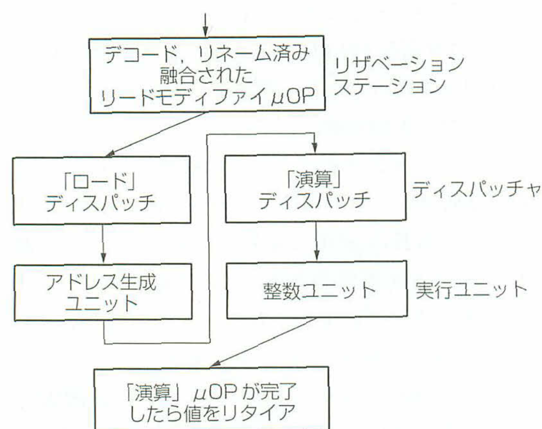


図35　Pentium Mの融合されたリードモディファイの流れ

れる．これらの実行は独立して行われ，融合されたストア命令のリタイアは，両方の操作が完了した時に発生する(**図33**)．

融合されたリードモディファイ命令を形成する2個のμOPは，アドレス依存があるため，逐次的に適切な実行ユニットに発行される．ロード操作のディスパッチは，そのソースオペランド(ベースやインデクスレジスタ)が用意できたときに実行される．演算操作は，ロードが完了し，もう一方のオペランドの用意ができたときに実行される．融合されたリードモディファイ命令のリタイアは，両方の操作が完了したときに発生する(**図34**)．

上述の論文にはとくに明記されていないが，x86の特徴であるリードモディファイライト命令はリードモディファイ操作とストア操作の組み合わせなので，1個の融合されたμOPとしてリザベーションステーションに登録され，ディスパッチ時に，3個または4個のμOPに分解されるのであろう(メモリのアドレスが同一なのでアドレス計算が1回省略できる)．

インテルによると，融合されたμOP構造はアウトオブオーダロジックで処理されるμOPの数を10%以上減少させることが判明している．μOPの数が減少するため，発行，リネーム，リタイアのスループットが増加し，結果的に性能を増加させる．とくに，命令デコードに監視/複雑デコーダが不要になるため，プロセッサのデコード，割り当て，リタイアのバンド幅を3倍に拡大するとしている．

μOPフュージョンによる性能向上は，典型的な例では整数コードで5%，浮動小数点コードでは9%である．ストア操作の融合は，とくに整数コードの性能向上に寄与する．浮動小数点コードの性能向上は，ストア操作とリードモディファイ操作(**図35**)の両方の形式が寄与するという．

● Dedicated Stack Engine

IA32はCISC命令であり，PUSH，POP，CALL，RETなどスタック操作を多用する．これらの命令はスタックポインタ(ESP)の値をアドレスとして使用する．このため，データの移動とは別にスタック計算用のμOPが余分に発行され，μOPの命令数が増加する．また，ESPが更新されないと次のPUSHやPOPが実行できないという依存性も発生する．従来は複数のスタック操作命令を同時にデコードしようとしても，ESPの値が確定していないため，それが不可能だった．Pentium Mでは，命令デコーダの近くに専用回路を設けることで，非常に効率的にESPのこれらの副作用を扱えるようになっている．

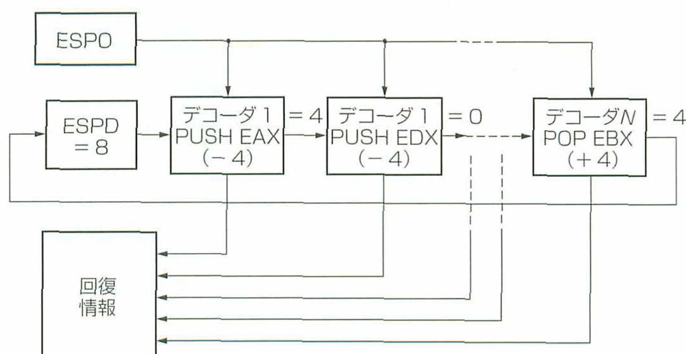


図36　Pentium Mの専用スタックエンジンの論理

その基本原理は，プログラマに見えるESP(ESPP)は，アウトオブオーダ実行エンジンの中にあるESPレジスタ(ESPO)に差分(ESPD)を加えたものである．つまり，

ESPP = ESPO + ESPD

であり，ESPDは命令デコーダで管理できる．つまり，前の命令でのESPの変化量は±4であることが多い(PUSHA，POPAは例外)．命令デコード時にESPDからの変化量を計算し，ESPPの値を推定することで複数命令の同時デコードを可能にするものである(図36)．この場合，ESPDの更新は専用の加算器で行う．

この操作は，ESPをデスティネーションオペランドとする命令には効果がない．この場合は，ESPPを計算するμOP(ESPOとESPDを加算)を余分に追加してESPOが更新されるのを待つ．その後はESPDを0とみなしてデコードしてよい．もともとESPDが0の場合は，このような同期化処理は不要である．

また，Pentium Mは投機実行を行うので，分岐予測が外れた場合はESPOやESPDの値をある時点まで巻き戻さなければならない．ESPOはアウトオブオーダ実行エンジンのレジスタの一部なので自動的に回復される．ESPDに関しては，その値を保持するテーブルを用意して対応する．

Dedicated Stack Engineを搭載することで，ESPを同期化するμOPを挿入したとしても，μOPの数が5%減少するという．それよりも，命令デコードのバンド幅が向上したことが性能に大きく寄与するらしい．また，消費電力も5%程度の削減になるという．

15 Hammerのパイプライン

● Quanti Speedアーキテクチャを採用

Athlonは，AMDがP6(Pentium Ⅱ)への対抗として開発した32ビットMPUである．そして，HammerはAthlonの後継にあたる64ビットMPUである．マイクロアーキテクチャの基本構造は，HammerとAthlonではよく似ている．だが，Hammerでは，I/Oやマルチプロセッサ接続を行うHyperTransportを3ポートとNorth Bridge(DDR SDRAMコントローラ，APICなど)を内蔵している点が異なる．

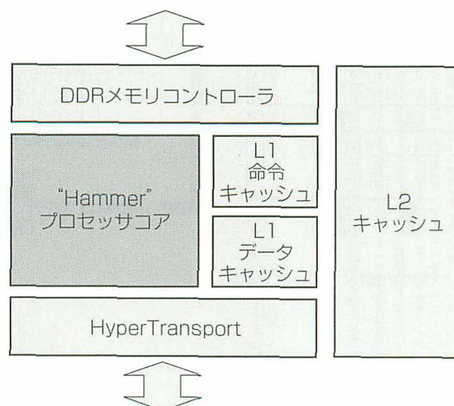


図37　Hammerの構成

図37にHammerの構成を示す．キャッシュとCPUコアが分離され，いろいろな構成(高級版や廉価版など)に対応できるようになっている．Hammerアーキテクチャを採用するMPUとして，OpteronブランドのSledgeHammer(サーバ，EWS向け)とAthlon64ブランドのCrawHammer(PC向け)が発表されている．これらにどのような相違があるのかは不明である．しかし，L2キャッシュの容量，DRAMコントローラやHyperTransportのバス幅/ポート数の違いによって区別されるのだろうと予想されている．

Hammerのブロック図を図38に示す．この図でL1キャッシュ，L2キャッシュ，TLB，分岐予測機構，North Bridge以外の部分がプロセッサコアである．

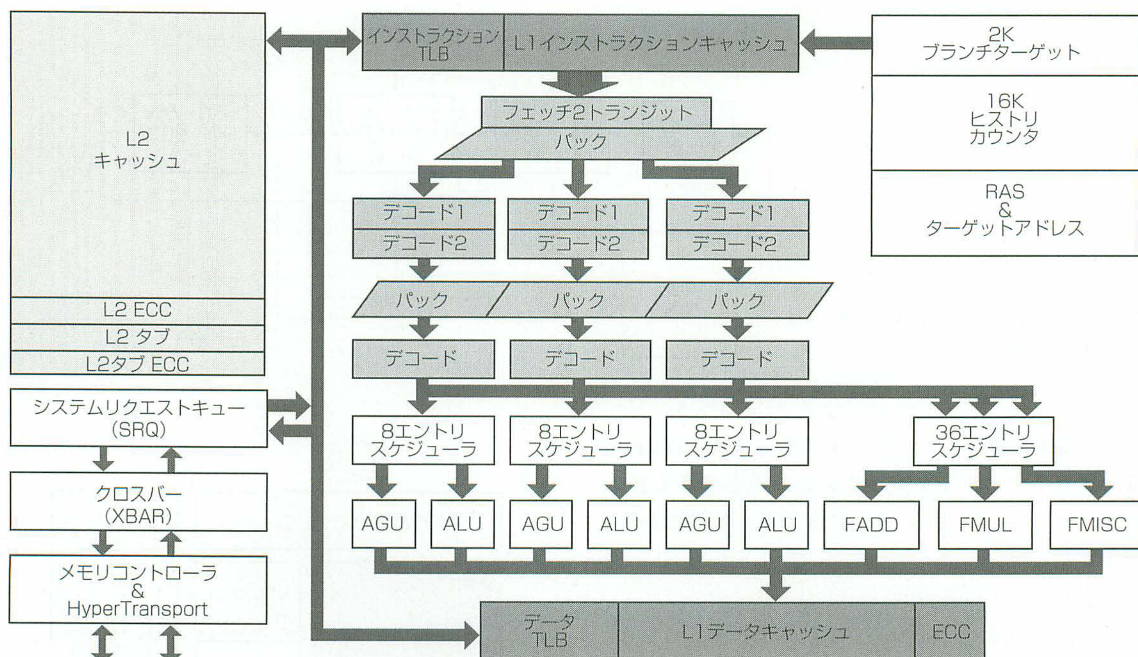


図38　Hammerのブロック図

なお，APIC(Advanced Priority Interrupt Controller)とシステム要求キュー(SRQ：System Request Queue)は二つのCPUを扱える構成となっている．頭初の発表によるとHammerはシングルCPUコアとなっていたが，実際には2，4，8CPUのCMP(Chip Multi Processor)構成が可能である．

ところで，HammerはAthlonと同様に，QuantiSpeedアーキテクチャを採用する．つまり，

- 9命令同時発行，スーパースカラ，パイプライン化されたマイクロアーキテクチャ
- 複数の並列 x86命令デコーダ
- パイプライン化された三つのスーパースカラ浮動小数点ユニット(FPU)
- パイプライン化された三つのスーパースカラ整数演算ユニット(ALU)
- パイプライン化された三つのスーパースカラアドレス生成ユニット(AGU)
- 72エントリの命令制御ユニット
- ハードウェアによるデータのプリフェッチ(Hammerのブロック図にはない)
- 排他的，投機的に入れ替えを行うTLB
- 動的な分岐予測

により，IPCを向上させている．同じくQuantiSpeedアーキテクチャを採用するAthlon(**図39**)と，基本的には変わらない．しかし，整数演算系のスケジューラ(リザベーションステーション)が18エントリから24エントリに増加している．Athlonのスケジューラの実態は，3×6エントリに分割されているという．その意味で，HammerではALUとAGUのペアごとに2エントリの増加になっている．

しかし，AthlonとHammerの大きな違いは命令のフェッチ系にある．要するに，分岐予測機能の高度化とマルチプロセッサで高性能をねらっている(動作周波数の向上は当然)のだ．命令デコーダ自体に大きな変更はないようである．

● 強化された命令デコーダ

QuantiSpeedアーキテクチャは命令デコーダが強化されているのが特徴の一つである．正確にいえば，ドキュメントによっては，命令デコーダはQuantiSpeedの特徴には入っていない．しかし，Athlonで強化された命令デコーダはHammerでもそのまま受け継がれているようなので，ここではQuantiSpeedに含めておく．

IntelのP6(Pentium Ⅱ/Pentium Ⅲ)アーキテクチャまで，デコーダは対称的ではなかった(NetBurstでは命令トレースキャッシュを使用するので事情が異なる)．しかし，QuantiSpeedでは対称的なデコーダを備える．これは，命令をデコードする効率に係わる．

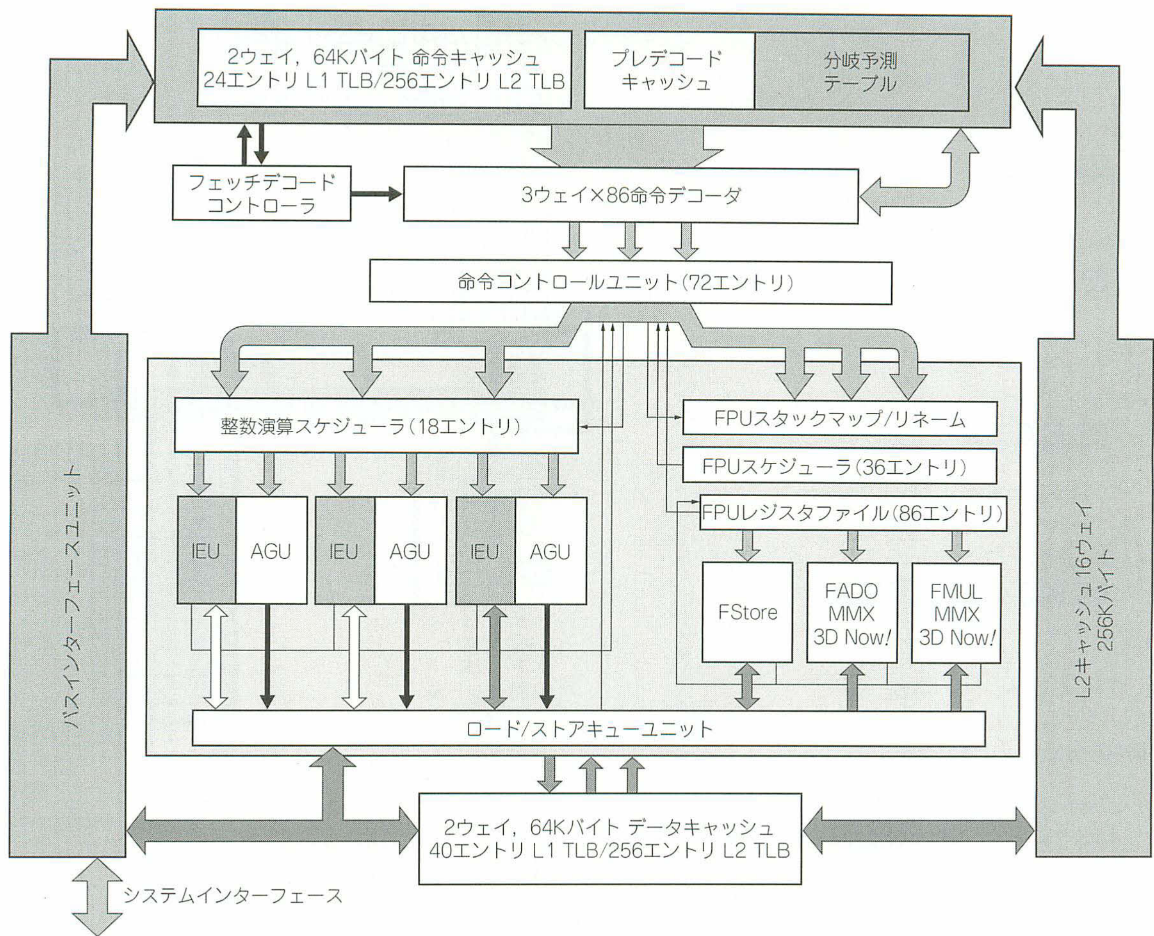


図39　Athlonのブロック図

P6とQuantiSpeedは，どちらも，複数の命令を一度にデコードするために，複数の命令デコーダを用意している．P6では二つの単純デコーダと一つの複雑デコーダで役割を分担している．一方，QuantiSpeedでは，同じ機能の命令デコーダ(ある程度複雑な命令をデコードできる)が3組対称に並べられている．

命令の分類としては，単純な命令，ある程度複雑な命令，非常に複雑な命令に分類できる．命令の整列がうまくいっており，それらが単純な命令であれば，1クロックで3命令をデコードできるのは同等である．しかし，複雑な命令が混じる場合は事情が異なる．

P6では，単純デコーダで処理できない命令は複雑デコーダに渡される．複雑デコーダでもデコードできない命令はマイクロコード命令シーケンサでデコードされる．命令デコードはインオーダに行われるので，複雑な命令が混じる場合は，1クロック間で，1命令あるいは2命令ずつしかデコードできず，スループットが低下する．それゆえ，P6では，レジスタ-レジスタ間演算といった単純な命令を使用しないと性能が出ないといわれている．それに対しAthlonでは，1世代前のPentiumやK6アーキテクチャに最適化したコードでも，それなりの性能が出るとされている．

QuantiSpeedでは，単純な命令とある程度複雑な命令を処理する直接径路(Direct Path)と，複雑な命令を処理するベクタ径路(Vector Path)に分かれて並列にデコードを行う．これは，AthlonではSCANステージ，Hammerではピックステージで選択される．

直接径路ではx86命令をデコードして，1クロック間に三つのMacroOPを出力する．このMacroOPは，後のアーリデコードあるいはパックステージで結合され，RISCライクなμOPに変換される．ベクタ径路ではマイクロコードROMがアクセスされ，1クロック間に最大3命令のMacroOPを出力する．これらが専用デコーダでμOPに変換される．

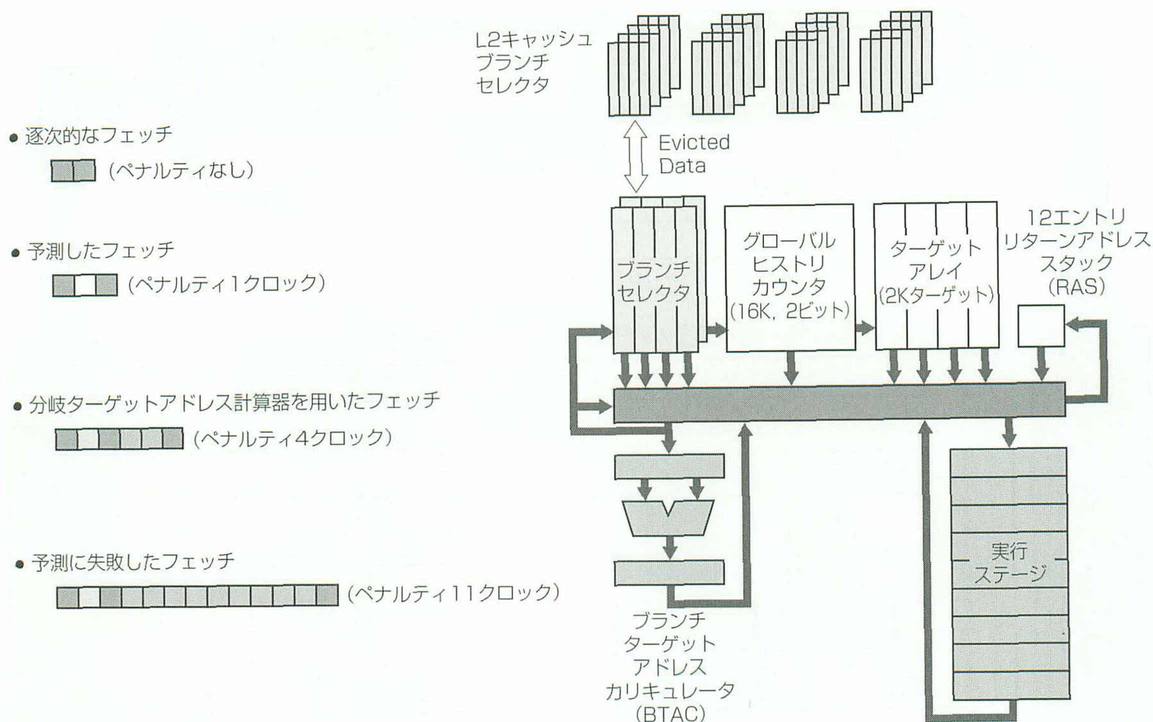


図40　Hammerの分岐予測

上述のように，QuantiSpeedの命令デコーダでは，見かけ上は直接経路とベクタ経路の違いをなくしている．これは，複雑な命令が混じっていても命令デコードのスループットが低下しないことを意味する．

● 分岐予測

投機実行が当たり前になっている最近のMPUでは，性能向上のために分岐予測機能はとくに重要である．AthlonとHammerの分岐予測機構は，Athlonに分岐ターゲットアドレス計算器がない点を除けば，分岐予測テーブルのエントリ数の違いはあるものの基本的には同じである．ここでは，Hammerの分岐予測について説明する．

Hammerの分岐予測は，図40に示すように3種類のテーブル(キャッシュ)とターゲットアドレス計算器から構成される．

- 大域履歴カウンタ(Global History Counter)
  分岐の方向(分岐/不分岐)を記憶．
  2ビット×8Kエントリ(Athlonの4倍)
- 分岐ターゲットアドレス配列(Branch Target Address Array)
  2Kエントリ(Athlonと同じ？)
- リターンアドレススタック
  (RAS：Return Address Stack)
  12エントリ(Athlonと同じ)
- 分岐ターゲットアドレス計算器(BTAC：Branch Target Address Calculator)
  1個(Athlonにはない)

フェッチした命令が分岐命令であるか否かは，L1キャッシュ格納時にプリデコードされて格納されている分岐選択情報(Branch Selector)によって判別される．そして，分岐命令をフェッチすると大域履歴カウンタ(GHC)と分岐ターゲットアドレス配列(BTAA)を参照し，前者から分岐方向を，後者から分岐先アドレスを得る．分岐先アドレスが判明すれば，そこから投機的に実行を始める．

なお，分岐選択情報はL2キャッシュにも存在する．これはL1キャッシュからL2キャッシュへ追い出す場合に，L1キャッシュの情報をECC領域に書き込むようだ．さすがに，L2キャッシュへのリフィル時にプリデコードを行うには回路が複雑になり過ぎた．

ただ，命令キャッシュでL1キャッシュからL2キャッシュへの追い出しが存在するのかという疑問もあるが，2001年のMicroprocessor Forumでは，そのように解説されていたらしい．図40においても，Evicted Data(立ち退かされたデータ)という表現がある．ということは，L1キャッシュとL2キャッシュは排他的

ステージ	分類	詳細	説明
1	フェッチ	フェッチ1	命令をL1キャッシュよりフェッチ
2		フェッチ2	
3		ピック	命令の整列
4		デコード1	x86命令をデコード
5		デコード2	中間コードに変換？
6		パック	個々の命令をパック
7		パック/デコード	命令をμOPに変換
8	実行	ディスパッチ	μOPをスケジューラ発行
9		スケジュール	スケジューラに格納
10		AGU/ALU	命令実行/アドレス生成
11		データキャッシュ1	データキャッシュアクセス
12		データキャッシュ2	
13	L2キャッシュ	L2リクエスト	
14		L2タグへのアドレス	Address to North Bridge
15		L2タグ	Clock Boundary
16		L2タグ，L2データ	SRQ Load
17		L2データ	GART/AddrMap CAM
18		L2からのデータ	GART/AddrMap RAM
19		データキャッシュへのMUX	Cross Bar
20		L1へライト，データ供給	Coherence/Order Check
21	DRAM		MCT Schedule
22			DRAM Cmd Q Load
23			DRAM Page Status Check
24			DRAM Cmd Q Schedule
25			Request to DRAM Pins
26			DRAM Access
27			Pins to MCT
28			Through North Bridge
29			Clock Boundary
30			Across CPU
31			ECC and MUX
32			Write Data Cache

図41 Hammerのパイプライン

(Exclusive Cahce)になっていて，最新の情報をL1キャッシュに置くように，L1キャッシュとL2キャッシュで適宜エントリの置き換えが行われていることになる．これは，L1キャッシュのほうがL2キャッシュより短時間でアクセスできるためであろう．たしかにHammerのブロック図(**図38**)ではL1命令キャッシュとL2キャッシュの径路は双方向になっている．このあたりはAMDからの詳細な資料を待ちたい．

GHCはBTAAより多くのエントリを持つので，分岐命令の処理に次のような場合が考えられる．

- GHCにヒット，BTAAにヒット
  1クロックのペナルティ
- GHCにヒット，BTAAにミス→BTACを利用して分岐アドレスを計算
  4クロックのペナルティ(BTACの時間は2クロック)
- GHCにミス→分岐するか否かはパイプラインの実行ステージまで不明
  11クロックのペナルティ(実行ステージまでの時間は9クロック)

ここでいうペナルティとは，逐次的な命令フェッチと比べて分岐先をフェッチするまでに要するむだな時間のことである．これに1クロック加算したものが分岐命令の実行クロックといえる．HammerではBTACを追加したことにより，Athlonに比べるとGHCミス時のペナルティを大幅に減少させている．

● パイプライン

Hammerのパイプラインを**図41**に，Athlonのパイプラインを**図42**に示す．AMDの説明によると，Hammerのパイプラインは整数演算が12ステージ，浮動小数点演算が17ステージだという．**図41**は整数パイプラインで，浮動小数点パイプラインは明らかにされていない．Hammerでは，Athlonでは1ステージだったL1キャッシュアクセスが2ステージに分割されているのが特徴的である．デコードも2ステージかけて余裕をもたせている．明らかに，高い動作周波数をねらった設計である．それなのに，2ステージしか違わないというのは，1クロックに処理する論理を見直したためであろう．

Hammerのパイプラインで，パックステージを謎のステージとする説もあるが，MacroOP(中間コード)

ステージ	整数演算	共通処理		FPU演算	説明(整数演算のみ)
		Direct Path	Vector Path		
1		FETCH			L1命令キャッシュから命令をフェッチ
2		SCAN			命令のDirect Path/Vector Path選択
3		ALIGN1	MECTL		命令キューへ格納/マイクロROMアドレス
4		ALIGN2	MEROM		命令の整列/マイクロROMフェッチ
5		Early DEC	MESEQ		アーリーデコード/マイクロコードデコード
6		IDEC/Rename			μOPに変換し，レジスタリネーム
7	SCHED			STKREN	命令スケジュール
8	EXEC			REGREN	命令実行
9	ADDGEN			SCHEDW	アドレス生成
10	DCACC			SCHEDW	データキャッシュへアクセス
11				FREG	
12				FEXEC1	
13				FEXEC2	
14				FEXEC3	
15				FEXEC4	

図42　Athlonのパイプライン

の結合はAthlonでも行われていたので，筆者はそれほど重要な意味はないと考える．あるいは，NetBurst(Pentium4)の命令トレースキャッシュやCrusoeのCMS(Code Morphing Software)のように，そこでスケジューリング(並び替えや最適化)を行ってから，μOPに変換しているのではと想像することはできる．

Hammerのパイプラインにおいて，L2キャッシュにアクセスする場合はさらに8ステージ必要である．なお，HammerではL2キャッシュへのアクセスとDRAMアクセスをオーバーラップできる．これは，L2キャッシュミス時のリフィルを高速に実行できる効果がある．DRAMへのアクセス自体はL2キャッシュのヒット/ミスを待たずに始まるようである．

16 Alpha21264

Alphaアーキテクチャは最初DECが開発し，その後Compaqに吸収された．そしてIntelに譲渡されることによって事実上消滅した．AlphaというMPUは，あと1～2製品開発されるらしいが，既に過去のものとなってしまった．2002年4月にCompaqは21364(EV7)を搭載するサーバ「Marvelous」を発表した．ISSCCではEV79(EV8とも)の発表もあったが，事実上，21364が最後のAlphaチップとなった．

しかし世界最高速のMPUとしてギネスブックにも掲載されたAlphaチップに言及しないのはどうも物足りない．ここでは，2001年時点での最新MPUであるAlpha21264について触れよう．

● Alphaのブロック図

図43にAplha21264のブロック図を示す．BOXという呼称はDECの伝統らしい．ここでは触れないが，StrongARMのブロック図でもIBOXとかEBOXという表現が散見される．

- 命令フェッチ，発行，リタイアユニット(IBOX)
- 整数演算およびアドレスユニット(EBOX)
- 浮動小数点演算ユニット(FBOX)
- 内蔵キャッシュ
  (命令キャッシュとデータキャッシュ)
- 外部キャッシュおよびシステムインタフェースユニット(CBOX)
- メモリ参照ユニット(MBOX)

Alpha21264のパイプラインのタイミングを**図44**に示す．この図を基にパイプラインの動作を説明する．

●ステージ0：分岐予測とライン予測を使用した命令フェッチ

プログラムの順序で命令キャッシュから最大4命令がフェッチされる．同時に，分岐予測テーブルと分岐履歴アルゴリズムを使用した分岐予測が行われる．Alphaには分岐予測機構とは別にライン予測機能がある．命令キャッシュの中にライン予測領域があり，この領域の情報にしたがって次にフェッチするキャッシュラインを選択する．

フェッチ予測器の目的は，分岐予測がTAKENと判断されるときに生じるパイプラインバブル(分岐先をフェッチするまでのむだ時間)を削除するためである．つまり，予測が当たれば，分岐先の命令をペナル

命令キャッシュ
Ibox
フェッチユニット
VPCキュー
仮想アドレス
ITB
Nextアドレス
分岐予測
命令
物理
アドレス
プリデコード
リタイヤ
ユニット
デコード&
レジスタリネーム
128
整数演算イシューキュー(20エントリ)
FPイシューキュー
(15エントリ)
Cbox
プローブ
キュー
デュプリケート
タグストア
IOWB
ビクティム
バッファ
アービタ
キャッシュ
データ
128
キャッシュ
インデックス
20
システム
バス
64
システム
アドレス
15
Ebox
Address
ALU0
(L0)
INT
UNIT
0
(U0)
INT
UNIT
1
(U1)
Address
ALU1
(L1)
整数演算レジスタ0
(80エントリ)
整数演算レジスタ1
(80エントリ)
Fbox
FP
ADD
DIV
SQRT
FP
MUL
FPレジスタ
(72レジスタ)
データ
128
Mbox
DTB
(デュアルポート128エントリ)
ロード
キュー
ストア
キュー
ミスアドレス
ファイル
物理アドレス
データ
デュアルポートデータキャッシュ

図43　Aplha21264のブロック図

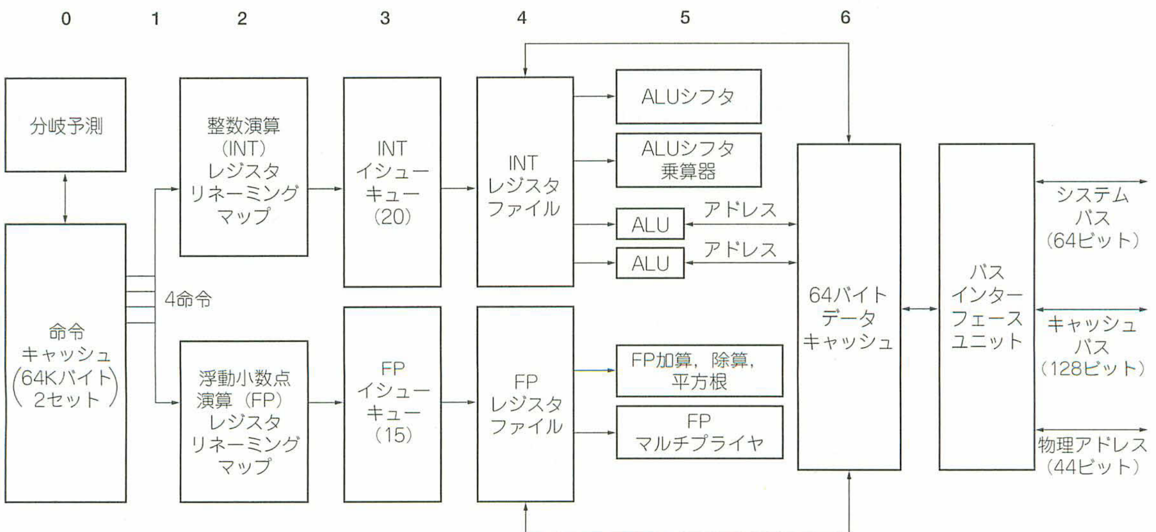


図44　Alpha21264のパイプライン

ティなしでフェッチできる．分岐予測とライン予測が異なった結果を返す場合は，分岐予測が優先される．ただし，条件分岐以外のコール命令とジャンプ命令では，ライン予測の方が優先される．

● ステージ1：命令スロットを形成し命令をリネーム(マップ)ハードウェアに転送

このステージでは命令キャッシュからフェッチした4命令を組み合わせ，一つの命令スロットにして，リネームハードウェアに転送する．ここで，命令が使用するハードウェア資源に基づいて，それぞれがどの演算器で実行されるかを決定する．これは，利用する資源を振り分けることでユニット間の負荷を低減する意味がある．演算器は上位と下位に分かれており，この時点で命令を振り分けてしまう．命令には，上位のみで実行可能(U)，下位のみで実行可能(L)，両方で実行可能(E)の3種類がある．Eに分類される命令はスロット形成時にUかLに決定される．

● ステージ2：レジスタリネーム(マップ)

このステージでは命令スロット内の各命令に対してレジスタリネームを行う．また，各命令は一意な8ビットの数値(inumと呼ばれる)が割り当てられ，その数値によってマップからリタイアまで(これが飛行中)の命令とプログラムの順序を識別する．つまり，inumはプログラムでの命令の順序を表す．マップされた命令とそれに対応するinumは，マップステージの終わりで整数または浮動小数点の命令キューに入れられる．

● ステージ3：命令キューから命令を発行

20エントリの命令キュー(IQ)は1サイクルに4命令を発行する．15エントリの浮動小数点キュー(FQ)は1サイクルに2命令の割合で，浮動小数点演算命令，条件分岐命令，ストア命令を発行する．このことから，レジスタリネーム論理とパイプラインの終端の間で最大80命令が「飛行中」になる．

● ステージ4：レジスタリード

命令キューから発行された命令は，オペランドデータを整数または浮動小数点レジスタファイルからリードし，バイパス(フォワーディング)されたデータを受け取る．

● ステージ5：整数演算と浮動小数点演算の実行

EBOXとFBOXのパイプラインが実行を開始する．つまり，前のステージでレジスタファイルからリード(またはバイパス)されたデータを処理する．

● ステージ6：データキャッシュアクセスあるいは演算結果の格納

ほとんどの整数演算命令はこのステージでレジスタに結果を書き込む．浮動小数点命令は，パイプライン的に処理され，所定のサイクルを経てレジスタにライトする(スループットは1サイクル)．

メモリ参照命令はデータキャッシュとデータ転送バッファにアクセスする．通常，ロード命令はタグ部とデータ部にアクセスするが，ストア命令はタグ部のみにアクセスする．ストアするデータはストアキューに書き込まれ，ストア命令がリタイアするまで保持される．これは投機実行中のストアの結果をライトさせないようにするためである．ロードは読み捨てればいいが，ストアでキャッシュやメモリを変更したら取り返しがつかない．

まとめ

以上，実際のプロセッサのスーパースカラ構造について解説してきた．

たとえば，SH-4のパイプラインに関しては資料がほとんど存在しない．しかし，x86系に関しては資料は豊富である．そういう意味では，SH-4は筆者の想像，x86系は既存の資料の受け売りが多いことは否定しない．多少の誤りはご容赦願いたい．

文章の量はかなり多くなったが，内容的にはそれほど難しいことは述べていない．最近のMPUはアウトオブオーダなスーパースカラが常識のようになってきているので，スーパースカラの基礎をおさえておくことは必須であろう．

第4章

キャッシュ構造の違いから，680x0/i486/R4000のキャッシュの動作まで

キャッシュのメカニズム

一口にキャッシュといっても，フルアソシアティブ/ダイレクトマップ/2ウェイセットアソシアティブなどのライン選択方式，ライトスルー/ライトバックの書き込み制御方式，LRU/FIFO/ランダム方式といったリプレースメント方式など，キャッシュの構造や動作でさまざまな違いがある．ここでは，それぞれのキャッシュ方式の違いを詳しく解説する．

その昔，フォン・ノイマンがプログラム内蔵方式，つまりプログラムもデータと同じようにメモリ中に格納する方式を提唱して以来，その方式は現在のコンピュータアーキテクチャの基本理念となっている（フォン・ノイマンがプログラム内蔵方式の提唱者というのは正確には誤りだが，ここでは通例にしたがっておく）．PentiumにしろPowerPCにしろ，現在でもこの方式から脱却してはいない．当然のことながら，ほとんどすべてのMPUは，プログラムを実行するときにはメモリへアクセスしなければならない．そして，そのメモリへのアクセス時間がプログラムの実行性能にも影響を与えてしまう．これが「フォン・ノイマン・ボトルネック」と呼ばれる現象である．MPUの性能向上のためのキーポイントの一つはフォン・ノイマン・ボトルネックの削減にあるといっても過言ではない．

● キャッシュメモリとは？

フォン・ノイマン・ボトルネックを削減するための手っ取り早い方法は，高速な（アクセス時間の短い）メモリを使用することである．世の中にはいろいろな種類のメモリ（記憶装置）があり，アクセス時間に応じて**図1**のようなメモリ階層を形成している．高速なメモリは高価であるため，大容量で使用することは難しい．そこで，**キャッシュメモリ**という構造が用いられる．

キャッシュ（cache）とは「隠し場所，貯蔵所」という意味で，キャッシュメモリとは原則としてプログラムで意識する必要のない高速な隠しメモリのことである．具体的には**図2**のように，小容量で高速なキャッシュメモリと，大容量で低速なメモリを階層構造に組み合わせる．

動作としては，低速メモリ（大容量）の内容の一部をキャッシュメモリ（小容量）にコピーしておき，MPUは，通常はキャッシュメモリのみをアクセスする．アクセスすべき内容がキャッシュメモリにない場合は，低速メモリの内容をキャッシュメモリへコピーし直し，そこをアクセスする．このときは低速なメモリからのコピーが発生するので多少時間がかかるが，2度

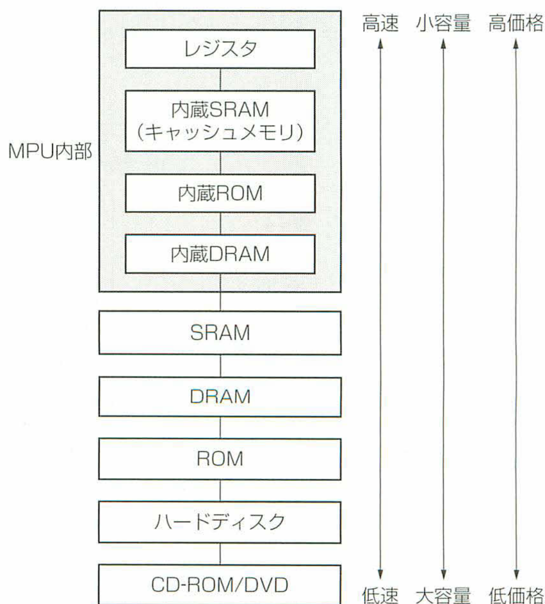


図1　メモリの階層

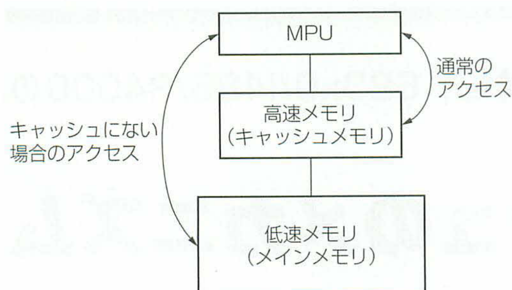


図2　キャッシュメモリの構造(概念図)

目以降はキャッシュだけにアクセスするので高速となる．たとえばプログラムがループ処理をする場合や，同じ変数を何度も読み書きするような場合，キャッシュへコピーされた命令やデータにアクセスすることになるので，プログラムが高速に実行されるというわけだ．これは，プログラムのメモリアクセスには局所性があるという経験則が基本原理となっている．

キャッシュメモリは単に「キャッシュ」と呼ばれることが多い．本章でも，以下ではキャッシュと表記する．また，低速メモリからキャッシュへコピーのし直しはリフィル，またはリプレースと呼ばれる．

● 昔は外付けSRAMで，現在はMPU内蔵で

現在ではMPUにキャッシュが内蔵されることは珍しくない．しかし，LSIの集積密度がそれほど高くなかった10年くらい昔は，SRAMを使用してMPUの外部にキャッシュを構成していた．とはいえ，SRAM自体が非常に高価だったため，本当に性能の必要な大型計算機などでしかキャッシュは採用されていなかった．ところが，現在主流のRISCではキャッシュの存在が前提で，メモリへのアクセスは，とりあえずキャッシュヒットするものと仮定してアーキテクチャが決定されている．LSI製造技術の進歩には目を見張るものがある．

なお，本章ではMPUに内蔵されているキャッシュ，とくに1次キャッシュを念頭において解説しているが，解説そのものはキャッシュについての一般論である．

1 キャッシュの内部構成

● キャッシュの構成

キャッシュは，高速，(比較的)小容量である点を除けば通常のメモリと変わりはない．アドレスを与えると対応するデータが出力される．ただし，低速なメモリ(メインメモリ)の一部をコピーしたものなので，対応するアドレスのデータが格納されていないことがある．これを**キャッシュミス**(あるいはミスヒット)という．このキャッシュミスを検出するため，特殊な構造を採用している．具体的には，タグ部とデータ部と呼ばれるメモリの組(これをラインまたはエントリと呼ぶ)の集合がキャッシュである(**図3**)．各アドレスに対して特定のラインが選択され，そのラインのタグ部の内容が与えられたアドレスに一致すればヒットであり，そのラインのデータ部の内容が与えられたアドレスの内容である(有効)ことがわかる．

逆に，タグ部の内容が与えられたアドレスに一致し

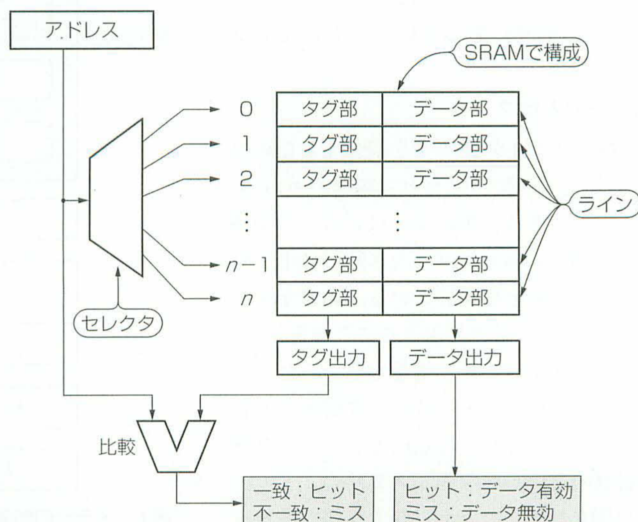


図3　キャッシュの内部構成

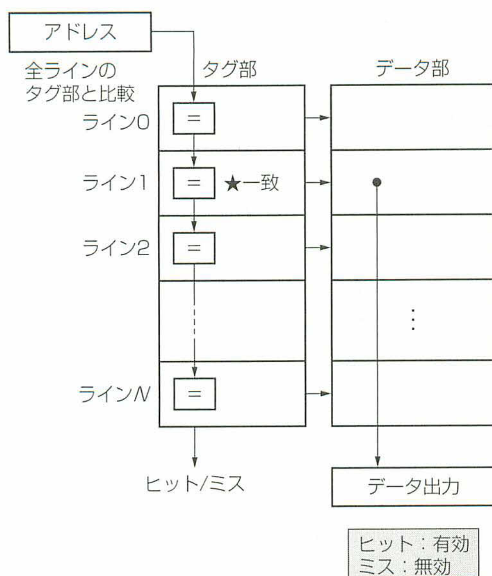


図4 フルアソシアティブ方式

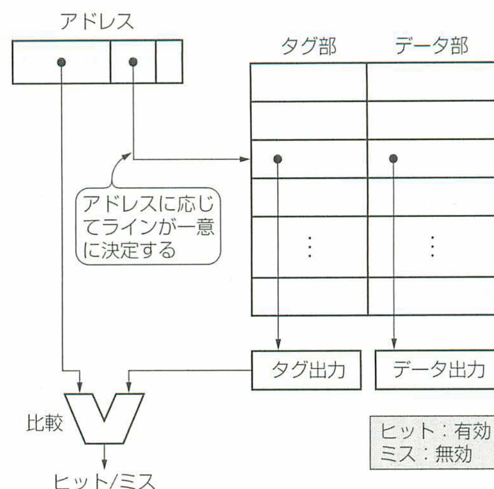


図5 ダイレクトマップ方式

なければミスであり，データ部の内容は与えられたアドレスのものではない(無効)．現実にはタグ部の中には，ラインの内容が有効なものであるか否かを表す「バリッドビット」も含まれている．バリッドビットが無効を示していれば，アドレスとタグが一致してもミスとみなされる．

また，データ部の容量はまちまちである．昔は，1ワード(4バイト)の場合が多かったが，現在では4ワード(16バイト)や8ワード(32バイト)が主流である．一般に1ラインのデータ部の容量(バイト数)が大きくなるほど，タグ部に必要なビット数を少なくできる．ただし，データ部の容量を大きくしすぎると，アクセスするアドレス範囲がランダムな場合にキャッシュのヒット率が低下し，性能が低下する．このため，データ部の容量の決定は，予想されるヒット率や利用できる回路規模(この場合は面積)を考慮して決定しなければならない．

● ラインの選択方式(連想方式)

キャッシュでは，アドレスが与えられるとある一つのラインが選択される．この方式には，大きく分けて次の3種類がある．

(1) フルアソシアティブ方式
(2) ダイレクトマップ方式
(3) nウェイセットアソシアティブ方式($n \geqq 2$)

● フルアソシアティブ方式

この方式の概念図を図4に示す．フルアソシアティブ方式において，与えられたアドレスはすべてのタグ部の内容と比較される．アドレスとタグが一致するラインが存在すればヒット，存在しなければミスである．図4の例ではライン1がヒットしているので，ライン1のデータ部の内容が有効なデータとして出力される．

この方式は直感的にわかりやすく，ラインをもっとも有効利用できる(したがって，同じライン数ではもっともヒット率が高い)方式であるが，全ラインのタグ部との比較のための論理回路が巨大になるため，また，後述するキャッシュミス時にリフィルするラインを決定するためのLRU(Least Recently Used)処理が複雑になるので，あまり採用されない．

もっとも，LRU処理をあきらめて，FIFO(First In First Out)制御やランダムな選択でリフィルするラインを決定することも考えられる．その場合，ネックとなるのはタグ部の比較論理の回路規模だけである．ライン数が少数(64程度)であれば，連想メモリなどを用いて比較回路を構成することは難しくない．そのため，この方式は，MMUのTLB(Translation Look-aside Buffer)において，仮想アドレスから対応する物理アドレスを選択する(アドレス変換)場合に採用されることが多い．

● ダイレクトマップ方式

この方式の概念図を図5に示す．この方式では，与えられたアドレスをデコードして特定の一つのラインに対応させる．デコードといっても大袈裟なものではなく，単にアドレスの1部分のビット列でラインを選

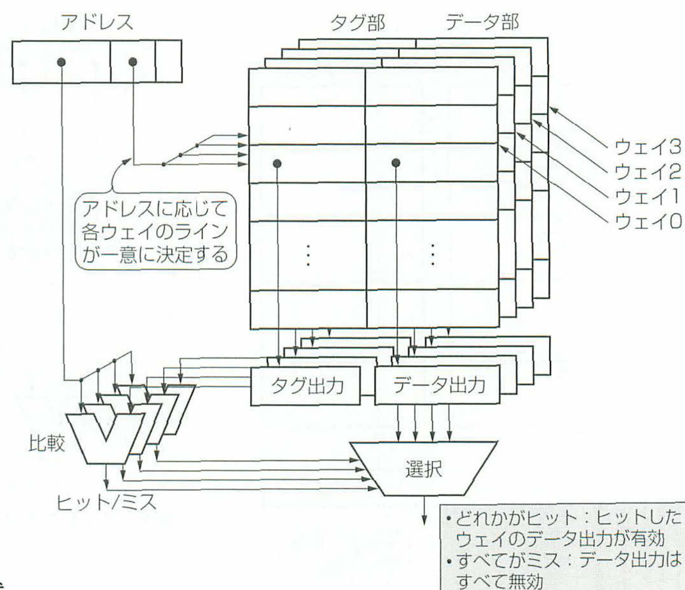


図6　4ウェイセットアソシアティブ方式

択することが多い．キャッシュの構成が256個のラインからなり，1ラインのデータ部が4ワード(16バイト)だとすれば，現在のMPUではバイトごとにアドレスが割り振られているので，アドレスのビット4からビット11の8ビットで参照するラインの番号を決定すればよい(8ビットなので256種類の値を指定できる)．

もっとも，アドレス内の連続する8ビットで指定した場合，アクセスするアドレス範囲が大きい場合はヒット率が低下する恐れもあるので，アドレスの上位数ビットを考慮したり，アドレスの二つの部分のビット列の排他的論理和を計算したりして参照するラインを決定する場合もある．

この方式は，キャッシュリフィル時のラインが一意に決定されるのでLRU制御を行う必要がなく，回路構成も単純なため(したがって高速に動作し，消費電力も少ない)，1世代前のMPUの内蔵キャッシュに多用されていた．

● *n*ウェイセットアソシアティブ方式

この方式の概念図を**図6**に示す($n = 4$の場合)．見てわかるようにnウェイセットアソシアティブ方式は，ダイレクトマップ方式の構成をn個並列に並べたものであり，それぞれが「ウェイ」と呼ばれる．n個のタグの比較器をもち，アドレスをデコードして決定される各ウェイに属するラインのタグ部出力を同時に比較する．一つでも一致するラインが存在すればヒットである．

この方式は構造が比較的単純で，ダイレクトマップ方式と比べてキャッシュのヒット率を上げることができる(最悪でもダイレクトマップと同じ)ため，もっとも多く採用されている．最新のMPUでは$n = 2$または4で構成されることが多いようだ．nの値を大きくすればするほどキャッシュのヒット率は向上するが，nが十分大きい場合はnと$n + 1$でのヒット率に大差はない．経験的には，$n = 4$が回路規模とヒット率を考慮した場合の最適解であるとされている．

なお，各ウェイに含まれるライン数が1で，nがラインの総数に等しい場合がフルアソシアティブである．nウェイセットアソシアティブ方式は，ダイレクトマップ方式とフルアソシアティブ方式の折衷案ということもできる．

ところで，IntelのStrongARM(XScale)は32ウェイセットアソシアティブと，驚異的なウェイ数を実現している．これは，ほとんどフルアソシアティブ並みといえる．ARMの文献を読むと，この32ウェイ構造は連想メモリによって実現しているそうである．そうなると，フルアソシアティブとどう違うのかという疑問が湧く．その実装方式は明らかにされていないが，どうやらフルアソシアティブキャッシュを32分割して，1ウェイ当たり64エントリ(キャッシュサイズ16Kバイトの場合)で制御しているようである(64エントリのフルアソシアティブキャッシュが32個あ

る）．連想メモリがタグの比較も行うので，1ウェイからは1ビットのヒット/ミス信号が出力されるのみである．これは32個のタグを同時に読み出すよりも効率がよさそうである．もっともこれは，仮想アドレスキャッシュ（詳細は後述）だからできる芸当であろう．

● 各方式でのキャッシュの効率

ただし，キャッシュのライン数（＝サイズ）が多いことがキャッシュ効率と直接には結び付かないことにも注意したい．同容量のキャッシュサイズの場合，連続的にキャッシュできるエリア，ないしはウェイごとのキャッシュ容量は，

　キャッシュ容量/n

で表される．

ここで，たとえば容量が0x800バイトのnウェイセットアソシアティブ構成のキャッシュを考える．$n=8$の場合，各ウェイの容量は0x100バイトである．1ラインの容量を16バイトとすると，アドレスのビット7～4（4ビット＝ライン数は16）が各ウェイのラインへのインデックスとなる．そして次のような3種類のアクセスパターンで，キャッシュの効率を見てみよう．

▶ アクセスタイプaの場合

さて，プログラムがアクセスするアドレスが，

　0x010，0x210，0x410，0x610，0x810，0xA10，
　0xC10，0xE10　　　　　…（アクセスタイプa）

というパターンで考えてみよう．これは，どれもラインへのインデックスは0x01であり，8ウェイあればすべてのアドレスをキャッシュできる．それでは，$n=4$の場合はどうだろう．各ウェイの容量は0x200バイトであり，アドレスのビット8～4（5ビット＝ライン数は32）がラインへのインデックスとなる．上の八つのアドレスに対して，この場合もインデックスはすべて0x01となる．したがって，4ウェイでは八つのうちの四つしかキャッシュすることができない．効率は半分に低下する〔図7（a）〕．

▶ アクセスタイプbの場合

次にプログラムがアクセスするアドレスが，

　0x010，0x110，0x210，0x310，0x410，0x510，
　0x610，0x710　　　　　…（アクセスタイプb）

であるとどうなるだろう．8ウェイの場合は，すべてのインデックスが0x01なので，先の例と同じく，すべてをキャッシュできる．一方，4ウェイの場合は，

　0x010，0x210，0x410，0x610

のアドレスに対するインデックスは0x01だが，

　0x110，0x310，0x510，0x710

のアドレスに対するインデックスは0x11である．インデックスが0x01と0x11のアドレスが4組あることになるので，4ウェイでもすべてのアドレスをキャッシュできる．この場合のキャッシュ効率は同じである〔図7（b）〕．

▶ アクセスタイプcの場合

さらに，アクセスするアドレスが次のように偏っている場合を考える．

　0x010，0x110，0x210，0x910，0xA10，0xB10，
　0x1010，0x1110　　　　　…（アクセスタイプc）

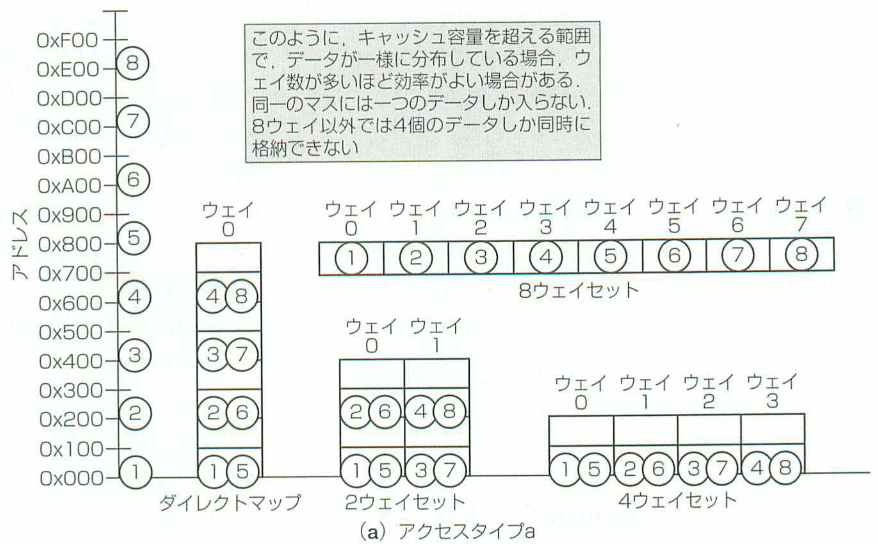
この場合は，大まかに2ヶ所にデータが分布している．上と同様に考えると，8ウェイでも4ウェイでも効率は変わらない．しかし，2ウェイだと少しだけ，そしてダイレクトマップとなると大幅に効率が落ちてしまう〔図7（c）〕．

以上の例でわかることは，ライン数よりもウェイ数を増やしたほうが効率的ということである．まあ，そのほうがフルアソシアティブ方式に近くなるので，当然といえば当然である．しかし，アドレスのばらつきがアクセスタイプbの組のような条件ならば，無理して複雑な8ウェイ構成にする必要はない．4ウェイで十分である．また，アクセスタイプcの組のような条件では，キャッシュ構成の複雑さとヒット率のトレードオフを考えると，2ウェイが最適といえる（2ヶ所に分布する傾向があるため）．

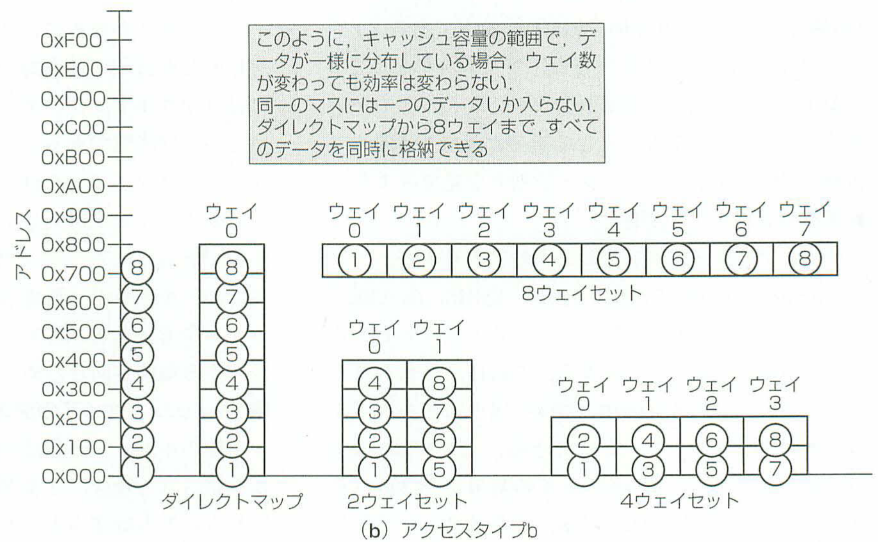
● キャッシュサイズの決定

実際のキャッシュ設計において，キャッシュサイズが限定される場合，さまざまなシミュレーションを行ってもっとも効率のよいと考えられるウェイ数に決定される．マルチスレッドで動作するプログラムをキャッシュする場合は，アドレスの下位ビットが一致する確率が高いので，ウェイ数を重視したほうが効率が上がる．Java処理系など，インタプリタやカーネルなどのある程度広がりをもった局所的な部分にアクセスが集中しがちな場合は，ウェイごとの容量が大きいほうが効率が良くなる傾向にある．キャッシュ構成の決定には，使用されるであろうOSやプログラムの種類などをよく吟味しなければならない．

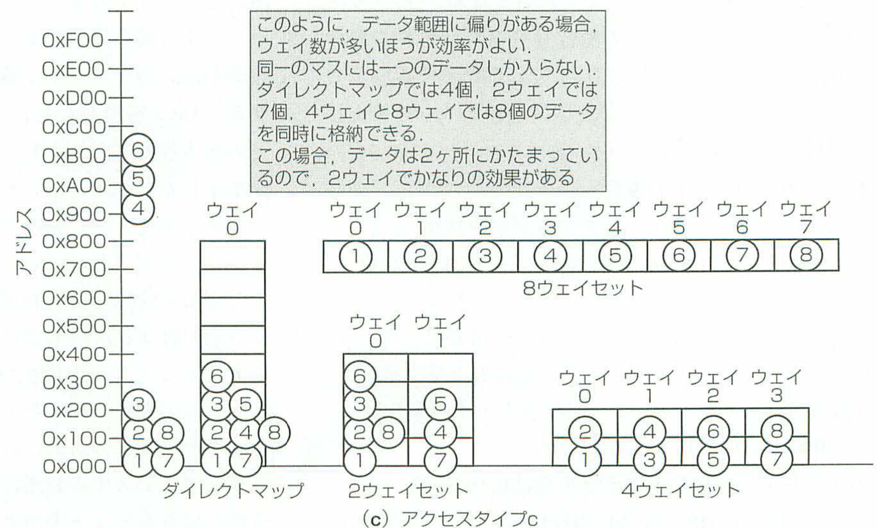
以上の性質を直感的にいえば，次のようになる．アクセスするアドレス範囲が真にランダムであれば，キャッシュのヒット率はキャッシュサイズのみで決定する．キャッシュの構成には無関係である．しかし，現実にはアクセスする範囲に偏りがあるので，ウェイに分けたほうがヒット率が上がる．たとえば，通常のア



(a) アクセスタイプa



(b) アクセスタイプb



(c) アクセスタイプc

図7　各方式でのキャッシュ効率の比較

プリケーションプログラムでは，命令はアクセスがユーザー領域とOS領域の2ヶ所に偏る傾向があり，2ウェイセットアソシアティブキャッシュが有効である．あるいは，データはプログラム固有のデータ領域とスタックの2ヶ所をアクセスするので，この場合も2ウェイセットアソシアティブキャッシュが有効である．しかし，現実にはプログラムの動きはもう少し複雑なものと考えられ，経験的には4ウェイセットアソシアティブがもっとも効率的とされている．そうであっても，構成の簡単さ，消費電力の考慮から，2ウェイセットアソシアティブ構成が採られる場合も多い．あるいは，キャッシュサイズが小さい場合は，アクセス範囲が十分ランダムとみなせるため，ダイレクトマップ構成も採用される．

2 キャッシュへのアクセス方式

キャッシュとは，アドレスを与えて(ヒットすれば)それに対応するメインメモリの(コピーしている)データを得るものである．この場合，与えるアドレスが仮想アドレスであるか物理アドレスであるかによって，特徴に若干の違いがある．

● 物理インデックス，物理タグ

この方式は，一般に「物理アドレスキャッシュ」と呼ばれる．物理アドレスからキャッシュのラインを決定し，出力されるタグ部には物理アドレスが格納されているものとして比較する．キャッシュをMPUの外部に取り付けるしかなかった昔は，MPUの外部バスから出力されるアドレス(もちろん物理アドレス)でキャッシュにアクセスするしか方法がないので，当然物理アドレスキャッシュである．次に述べる仮想アドレスキャッシュと違い，タスク切り替えごとにキャッシュを無効化する必要がないので，制御が簡単である．しかし，仮想アドレスから物理アドレスへのアドレス変換が終了しないとキャッシュにアクセスすることができないので，キャッシュのアクセス時間に余裕がなくなり，高速動作させることが難しいのが欠点である．

● 仮想インデックス，仮想タグ

この方式は，一般に「仮想アドレスキャッシュ」と呼ばれる．仮想アドレスからキャッシュのラインを決定し，出力されるタグ部には仮想アドレスが格納されているものとして比較する．この方式ではアドレス変換と同時にキャッシュにアクセスできるため，また，キャッシュ自身にタグ比較の論理を取り込むこともできるため，キャッシュアクセスに余裕ができ，高速で動作させることが可能である．しかし，欠点もある．メインメモリへの最終的なアクセスは物理アドレスで行われるので，メインメモリのデータは物理アドレスで一意に区別できる．つまり，物理アドレスが同じなら同じ場所，物理アドレスが異なれば異なる場所を指す．しかし，仮想記憶で動作している場合，仮想アドレスが同じでも，同じ物理アドレスを指し示しているとは限らない(ほとんどの場合，異なる物理アドレス)．ということは，単純に考えると，仮想アドレスだけでタグ比較を行っていると意図した物理アドレスと異なる場所からデータを取ってしまうことがある．これをエイリアシングまたはシノニムの問題という．

通常，仮想アドレスと物理アドレスの対応はタスクごとに決まっているので，タスクが切り替わるとキャッシュのタグ部に格納されている仮想アドレスは無意味なものになる．したがって，仮想アドレスキャッシュを採用する場合は，タスク切り替えごとにキャッシュの内容を無効化する必要がある．これは制御回路の増大を招く．これを防ぐ方法としてタグ部の中にタスクIDを一緒に格納しておき，タグの比較時に同時にタスクIDも比較することが考えられる．しかし，この場合はタグ部のビット数が増大する．また，ごく稀ではあるが，異なる仮想アドレスに同一の物理アドレスを対応させる場合もある．仮想アドレスキャッシュはこの場合に対応できない．

IntelのStrongARMは仮想アドレスキャッシュを採用している．最初のSA-110はタスクIDをサポートしていなかったが，これでは実用性に乏しいのか，Windows CEに採用されたSA-1100やSA-1110ではタスクIDをサポートするようになった．

● 仮想インデックス，物理タグ

この方式にはとくに決まった呼称はない(と思う)．仮想アドレスからキャッシュのラインを決定し，出力されるタグ部には物理アドレスが格納されているものとして比較する．これは，物理アドレスキャッシュと仮想アドレスキャッシュの折衷案である．アドレス変換と同時に仮想アドレスでキャッシュにアクセスし，アドレス変換が終了する頃に，キャッシュから出力される物理アドレスとアドレス変換した物理アドレスを比較する．そのためキャッシュのアクセス時間に余裕ができ，タスク切り替え時の無効化も必要ない．この方式はMotorolaのMC68040以降やMIPS系のMPUで採用されている．

3 リプレースメント方式

キャッシュはヒットすることが前提とはいえ，現実には頻繁にミスが発生する．この場合，キャッシュ内にメインメモリの新しいコピーをもってくる必要がある．このとき，どのラインに新しいデータを書き込むのかを決定する方法が**リプレースメント方式**である．書き込むラインが決定すれば，そこに新しいデータをリフィル(リプレース)する．ダイレクトマップ方式の場合は何の考慮も必要ない．アドレスに対して対象ラインは一つしかないので，そこをリフィルする．nウェイセットアソシアティブの場合は，与えられたアドレスに対して対象ラインはn個あるので，それから一つを選択しなければならない．フルアソシアティブの場合は，すべてのラインがリフィルの対象である．

● LRU(Least Recently Used)方式

この方式は，プログラムの(時間的な)局所性という経験則に依っている．すなわち，いちばん昔にアクセスされたラインはこれからアクセスされる確率が低いのでそこを更新する，というもっとも妥当な方式である．この方法では，nウェイセットアソシアティブ方式の場合は，各ウェイの同一インデックスにあるn個のラインに対するアクセス頻度の履歴を記憶しておく．そのために，$n=2$の場合は1ビット，$n=4$の場合は6ビット，$n=8$の場合は28ビットのメモリが必要である．フルアソシアティブの場合は全ラインのアクセス頻度の履歴を記憶しなければならないので，ほとんど非現実的なビット数のメモリが必要である．このため，LRU方式は，主としてnウェイセットアソシアティブ方式で用いられる．

この方式の欠点としては，ラインへのアクセス(ヒット)ごとにLRUメモリを更新しなければならないので，タイミング的に厳しいということくらいだろうか．

● FIFO(First In First Out)方式(ラウンドロビン方式)

この方式は，nウェイセットアソシアティブ方式に

Column キャッシュのヒット率に関して

キャッシュの目的はメインメモリが低速な場合，プログラムが意識しなくてもメモリアクセスが高速に行えるということである．つまり，メモリアクセスがキャッシュにヒットしなければ性能は低下する．いくら高性能なキャッシュメモリを使用してもヒット率が低ければ意味をなさない．そこで，キャッシュの構成にはヒット率を向上させるための仕組みが盛り込まれている．この章で述べてきたキャッシュの構成はヒット率を向上させる目的で(試行錯誤の末？)提案されてきたものである．ここで，キャッシュのヒット率を向上させるキーワードを明確にしておこう．

● 大容量

キャッシュメモリの容量は大きければ大きいほどヒット率が向上する．理想はメインメモリと同じ容量をもつことだが，それが実現可能ならキャッシュなどという特別な仕組みは不要である．

● 多ウェイ化

フルアソシアティブ方式がもっともヒット率が高い．回路構造上，現実困難な場合があるので，nウェイセットアソシアティブ構成で，ウェイの数をできるだけ多くするほどヒット率は向上する．

● リフィルサイズの増大

プログラムの性質にもよるが，キャッシュのアクセス時間(速い)とバスの転送スピード(遅い)の関係を考えると，一度に(バースト転送で)できるだけ多くのデータをキャッシュ内に取り込むほうがヒット率は向上する．アクセスするアドレス範囲が大きくなるほど，多数のラインをリフィルする必要が生じる．そのラインへのアクセス頻度が少なければ，多量のバースト転送がかえってバスネックになる恐れもあるので，リフィルサイズは大きければ大きいほどいいというものでもない．

StrongARMはリフィルサイズを8ワード(32バイト)としながらも，ライトバックは4ワードまたは8ワードのうち最適なほうを選択できるようになっている．つまり，1ラインについてダーティビットを，前半4ワード用と後半4ワード用の2ビットをもっており，リプレース時にダーティビットが1である4ワードのみをライトバックする．このようにしてライトサイクルのバス占有時間を低減している．この方式を採る理由は，1ラインすべてがダーティになる可能性は少ないという，DEC(本来の開発元)の主張による．

● LRU処理

データアクセスの(時間的，空間的)局所性が最大の拠り処である．ラインのリフィル時に，これからもアクセスする可能性の高いラインを書き潰していたのではヒット率は低下する．FIFO，ランダム方式に比べ，LRU方式のほうがヒット率が高い．

おいて，0，1，2，…，$n-1$，の順にリフィルするラインを決定するものである．キャッシュラインがすべて無効な状態からリフィルを続けていくと，ウェイは0，1，2，…，$n-1$の順にリフィルされていくので，この順に古いデータが格納されているとみなし，その順序で新しいラインを決定する方式である．アクセス頻度が無視されてはいるが，一応，古いラインからリフィルしていくという方針である．ヒットする場合に順序の更新が行われないので，当然LRU方式よりもヒット率は悪くなる．履歴の記憶に必要なメモリのビット数は，カウンタを形成すればいいので，$n=2$の場合は1ビット，$n=4$の場合は2ビット，$n=8$の場合は3ビットで足りる．LRU方式に比べて少ないビット数で済むのが特徴である．フルアソシアティブ方式の場合はラインの番号順にリフィルしていけばよいだろう．

先にも挙げたが，IntelのStrongARM(SA-1100)は，32ウェイセットアソシアティブという(嘘のような？)キャッシュ構成を採っているが，さすがにLRU方式ではなく，このFIFO方式を採用している．FIFO方式は，対象エントリの番号が順次回転していく(最後の次は最初に戻る)ので，ラウンドロビン(回転)方式ともいう．

● ランダム方式

この方式は，ランダム(無作為)にリフィル対象のラインを決定する方式である．どのアドレスも同じような頻度でアクセスされるはずという予測に基づいている．ラインを指定するために必要なメモリのビット数はFIFO方式の場合と同じである．1クロックあるいはキャッシュへの1アクセスごとにそのメモリを更新(たとえば+1)しておいて，リフィルが必要になった場合に，そのメモリの値が(たまたま)示しているラインをリフィルする．ヒット率としてはFIFO方式と大差ないと思われる．論理が単純なためか，この方式はけっこう多くのMPUで採用されているようである．

4 書き込み制御

キャッシュは何もリードするだけではない．書き込みを行う場合もある．キャッシュはメインメモリの内容をコピーしているものだから，常にメインメモリの内容と整合性(コヒーレンシ)が保たれている必要がある．それを実現するために，いくつかの制御方式が考案されている．

● ライトスルー(ストアスルー)方式

これはライトデータに関して，常にメインメモリにも書き込みを行う方式である．誰もが考えつく方式であろう．ライトアドレスがキャッシュにヒットする場合は，ライトデータをメインメモリと同時にキャッシュのデータ部にも書き込む．キャッシュミスの場合はキャッシュは無視してメインメモリのみにデータを書き込む方式が一般的である．

キャッシュミスの場合には，まずリフィルを行い，そのラインとメインメモリの両方にデータを書き込む方式もある．これはライトアロケートと呼ばれる．スタックなど，ライトしたアドレスは再びリードする傾向があるので，あらかじめそこのアドレスをキャッシュに入れておこうという発想である．ライトアロケートは，ライトしたアドレスを再びリードする確率が高くないと効果がない．ライトしたアドレスを再びリードする場合も，後で発生するはずのリプレースをライト時に先行して行うだけなので，トータルのリプレース回数には変化がない．この意味で，ライトアロケートが効果的かどうかという点については疑問が残る．

ライトスルー方式を採用する場合，ライトごとにメインメモリへの書き込みバスサイクルが発生するので，連続してライトを行う場合は，前の書き込みバスサイクルが終了するまで次の書き込みバスサイクルを開始できない．このときMPUのパイプライン処理が待ち合わせのために停止してしまう．それを防ぐために，ライトスルー方式を採用するMPUではライトバッファを数段分もっていることが多い．逆に，ライトバッファがないと性能が低下する．

● ライトバック(コピーバック)方式

この方式は，メインメモリへのライトアクセスを最小限に抑える方式である．つまり，ライトが発生しても(ヒットする場合は)キャッシュのデータ部のみしか更新しない．当然，メインメモリとの整合性は保たれなくなる．その代わり，そのラインの整合性が保たれていないことを記憶しておく．そして，後で一括してラインごとメインメモリに書き戻す．そのタイミングは，そのラインがキャッシュにミスし，新しいデータをリフィルしなければならないときである．ライトのいくつか(大半？)はキャッシュにヒットするので，メインメモリに対する書き込みバスサイクルの回数を削減することができる．このメインメモリへの一括した書き込み動作を特別に**ライトバック**と呼ぶ．

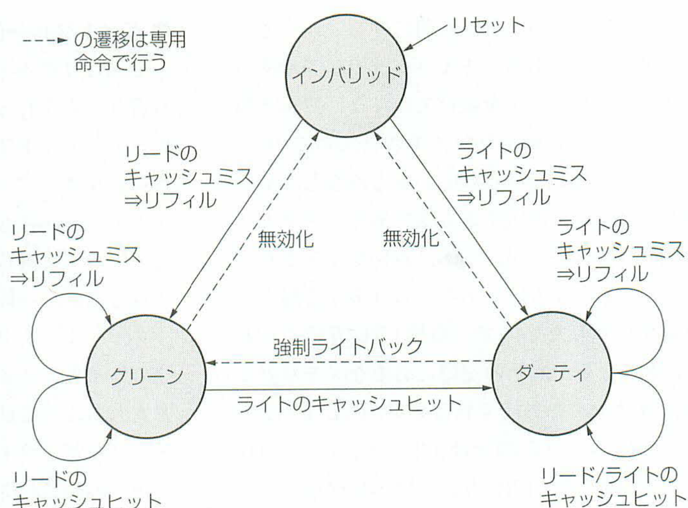


図8
ライトバックキャッシュの状態遷移

ライトバック方式のキャッシュはライトアロケートである．キャッシュミスが発生すると，まずリフィルを行って，そのラインのデータ部にライトデータを書き込む．このとき，メインメモリには書き込まない．また，ライトバック方式のキャッシュでは各ラインが現在のキャッシュ状態というものをもっている．メインメモリと整合性が保たれている状態をクリーン(Clean)，保たれていない状態をダーティ(Dirty)という．この状態を示す情報はタグ部に格納されている．**図8**にライトバック方式のキャッシュの状態遷移を示す．

ライトバックはライン単位で行われるのが一般的である．つまり，1ラインごとに1ビットのダーティビットをもって管理するわけだ．しかし，ラインのすべてがダーティになるのは稀である．たとえば，1ラインが32バイトだとすると，そのうち4バイト程度しかダーティにならないことがある．これを中途半端なダーティという．この場合，1ラインの32バイトすべてをライトバックするのは効率的ではない．本当にダーティな4バイトのみをライトバックできれば，ライトのバスサイクルが減少するので，メモリ効率が良い．これを実現するには，1ライン当たりのダーティビットを複数もつことである．たとえば，StrongARMは，8ワード(32バイト)の1ラインに対して，下位4ワード用と上位4ワード用の2ビットのダーティビットをもつ．キャッシュのリフィルは必ず8ワードで行われるが，ライトバックは，ダーティビットの状況に応じて，4ワードまたは8ワードで行われている．

5 キャッシュを支える各種機能

● リフィルサイズ

キャッシュミスが発生すると，そのラインはリフィルされる．通常リフィルはライン単位で行われる．たとえば，ラインのデータ部が4ワード(16バイト)なら，一度に4ワードのデータをメインメモリから読み込む．これは，いったんアクセスしたアドレスの近傍を再びアクセスする確率が高いという，またもやプログラムの局所性に依っている．また，キャッシュのリフィル時に発生するバスサイクルは一般にバースト転送と呼ばれるバスサイクルである．これは，メモリをバスクロック同期で連続的にアクセスする．最近のメモリデバイスはRAMにせよROMにせよページモードというモードをもっている(今流行のSDRAMも似たような動作をする)．このモードにおいて，最初のアクセスのアクセス時間はやや遅い(というか通常の速さである)が，連続するアドレスの2回目以降は，最初の半分程度のアクセス時間でアクセスできる．4ワードのデータを4回に分けてリードするよりも，4ワードのバースト転送を行ったほうがはるかに高速なのである(**図9**)．

MPUによっては複数のラインを同時にリフィルするものもある．これは，アクセスする可能性が高いアドレス範囲をあらかじめキャッシュに入れておくほうがヒット率の向上が見込めるためだが，ページモードとの相性のよさも考慮されているはずである．

現在のMPUでは，1回のリフィル時にリードする

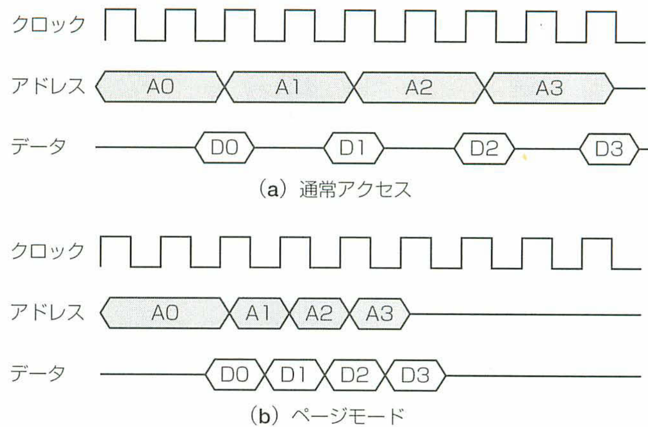


図9　バースト転送のイメージ

データ量(ラインのワード数，または，その倍数)は8ワードが多いようである．MPUによっては32ワード程度まで設定可能なものもある．プログラムの性質(分岐の発生頻度や同じアドレスをアクセスする確率の大小)を考慮しながら，最適な値をユーザーが設定できる．

頻繁に分岐が発生するプログラムでは，プログラムの実行はリフィルが終わるまで待たされるので，リフィルサイズが大きいとリフィル(＝キャッシュミス)の発生する確率が大きくなり(同じキャッシュラインへの分岐はしないと仮定)，命令実行が阻害されてしまう．後で述べるフェッチバイパスを行えば，性能低下は多少抑えられるが，リフィルの発生する頻度は同じなので，性能がバスネックになる．逆に，分岐がほとんど発生しないプログラムでは，できるだけ多くの命令をリフィルしたほうが得である(とくにフェッチバイパスを行う場合)．

かつて，筆者は某OS上でいくつかのアプリケーションプログラムの実行結果をトレースしたことがある．このとき，分岐命令の出現頻度は8命令に1回程度だった．この結果から，8命令分を1回にリフィルすればリフィルにかかる時間が最小になると考えられる．多くのRISCにおいて，命令長は32ビット(1ワード)なので，リフィルサイズは8ワードが最適ということになる．

● クリティカルワード

分岐先がキャッシュミスを起こす場合，その分岐先がキャッシュラインの先頭とは限らない．そんな場合，リフィルをキャッシュラインの先頭から行っていたのでは，目的の命令を取り込むまでに時間がかかってしまう．分岐先に対応するアドレスから先にリフィルしてほしい．当然ながらこういう考えが生まれる．分岐先が含まれる，キャッシュラインのワードをクリティカルワード(critical word ＝ 緊急にほしいワード)と呼び，クリティカルワードからリフィルを始める方式をクリティカルワードファースト(critical word first)と呼ぶ．

たとえば，4ワード(16バイト)のラインサイズを仮定し，分岐先が4番地であるとする．この場合，通常方式(シーケンシャル方式)は，0，4，8，12番地の順にリフィルするのだが，クリティカルワードファーストだと，4，8，12，0番地の順にリフィルする．もし，キャッシュラインのワードごとにバリッド(有効)ビットを備えるなら，最後の0番地の命令はリフィルしなくてもいいかもしれない．しかし，リフィルサイズを動的に可変とすると制御が複雑になるので，通常は行わない．

ちょっと考えればわかるが，クリティカルワードファースト方式ではフェッチバイパスを実行しないと意味がない．

ところで，クリティカルワードとよく似たリフィル方式にサブブロック(sub-block)方式がある．インタリーブ(interleave)方式ともいう．これは，クリティカルワードから始めて，それが属するブロックから順番にリフィルする方法である．具体的には，シーケンシャルなアドレスとクリティカルなアドレスの排他的論理和を計算して，リフィルを行う．つまり，4番地がクリティカルワードの場合，

0　XOR　4　→　4
4　XOR　4　→　0
8　XOR　4　→　12
12　XOR　4　→　8

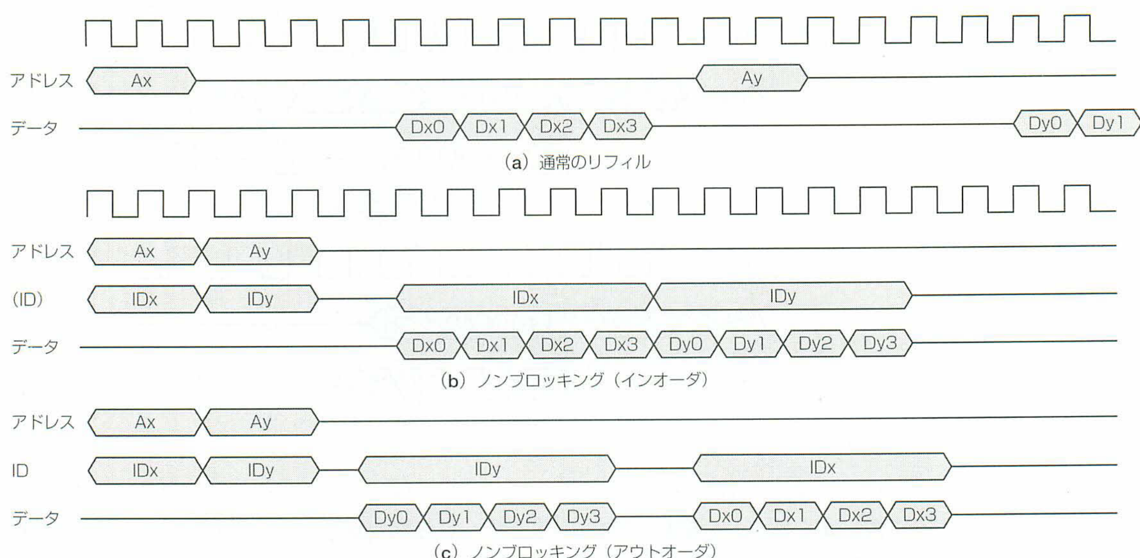


図10　ノンブロッキングキャッシュの概念

と計算されるので，4，0，12，8番地の順に命令を取り込む．これは，命令実行の効率を考慮したものではなく，メモリ側の効率を考慮したものである．クリティカルワードからアクセスする場合，次にアクセスするアドレスを加算なしで予測できるため，メモリからのデータ出力を，アナログ的に高速化できる．

● ノンブロッキングキャッシュ

通常，キャッシュミスが発生すると，リフィル動作(バースト転送)が終了するまでパイプラインが止まってしまう．ノンブロッキングキャッシュとは，キャッシュミスが発生してもパイプラインを停止せずに先に進める技術である．キャッシュミスをヒットのように扱うことから「ヒットアンダーミス」ともいう．

具体的な実装は，リフィルデータを格納するためのリードバッファを何組か用意しておき，キャッシュミスが発生するとリードバッファとリード(またはストア)を発生する命令を関連づける．リフィルはリードバッファに対して行い，キャッシュは暇を見て更新する．その間パイプラインを止めるようなことはしない．

さらに，リフィル要求と同時にバッファのID(番号)を同時に出力し，外部からはデータにそのIDを付けて返してもらう方式も考えられる．リフィルデータはIDで区別できるので，キャッシュミスを発生した順序でデータを返す必要はない(アウトオブオーダ)．もちろん，キャッシュミスを起こした順番にデータを返す(インオーダ)場合は，データを区別するIDは不要である．2次キャッシュをもつ場合や，マルチプロセッサ構成になると，アクセスごとにデータを用意できる時間が異なるので，アウトオブオーダなデータ応答は実効性能を上げる意味もある．**図10**にノンブロッキングキャッシュの概念図を示す．

ノンブロッキングキャッシュは，リードしたデータをすぐに参照しない場合に効果がある．このためにはコンパイラの命令スケジューリングによる最適化が必要になる．

リードしたデータを次に参照しない場合でも，連続するデータがバースト転送でキャッシュにリフィルされているので，ノンブロッキングキャッシュには後述するプリフェッチの効果も期待できる．

また，キャッシュミスはロード/ストア命令の実行に付随して発生する．そのため，ノンブロッキングキャッシュ環境下でのロード命令では，命令の追い越しが行われ，デスティネーションレジスタへの書き込みはアウトオブオーダになる．その意味で，ノンブロッキングキャッシュは複雑な制御となる．このため，ノンブロックキャッシュでは割り込み応答に時間がかかったり，例外発生時の正確さ(precise)を欠いたりする場合がある．これらを避けるため，普通はノンブロックキャッシュ機能を禁止するしくみがある．

個人的には，ロードしたデータの使用をそれほど先延ばしできるとは思えないので，ノンブロッキングキャッシュの効果については懐疑的である．しかし，某研究所のシミュレーション結果によれば，5段程度のバッファがあれば非常に有効という結果が出ているの

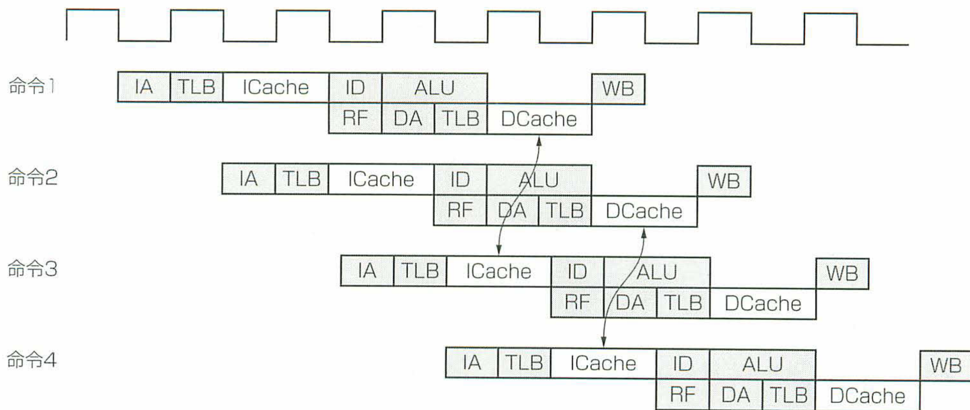


図11 R3000のパイプラインと命令キャッシュ/データキャッシュを参照するタイミング

で，もしかしたらそうなのかもしれない(多分，アウトオブオーダ方式の場合だろう).

● **命令キャッシュとデータキャッシュ**

図11にR3000のパイプライン動作を示す．この図で,「ICache」が命令キャッシュへの,「DCache」がデータキャッシュへのアクセスを示している．図を見ると命令1のデータキャッシュへのアクセスと命令3と命令4の命令キャッシュへのアクセスのタイミングが重なっている．命令キャッシュへのアクセスは毎回発生するが，データキャッシュへのアクセスはロード/ストア命令のみで発生するため，アクセスが重なることは多くないが，まったくないとはいえない．この場合，同じキャッシュからデータを参照することは(どちらかのアクセスを待ち合わせてパイプラインを一時停止しなければ)不可能である．R3000はパイプラインをできるだけ停止させないことを信条としているので，命令キャッシュとデータキャッシュを分けて独立なアクセスを可能にしている．このように，命令キャッシュとデータキャッシュをアクセスする経路を別々に設けるアーキテクチャを**(修正)ハーバードアーキテクチャ**という．米国ハーバード大学で初めて提唱されたのでこの名称があるのだろう．CISCではMotorolaのMC68020辺りで初めて採用されたように思う．

ハーバードアーキテクチャの欠点(?)は，命令キャッシュとデータキャッシュが別のため，命令書き換えに対応できないことである．また，同じアドレスの内容を命令とデータキャッシュでそれぞれ独立に記憶する場合があるので,メモリのむだといえばむだである．

逆に，命令とデータで同じキャッシュをもつのが**ユニファイドキャッシュ**である．Intelのi486あたりまでがこの方式を採用している．命令書き換えに対応できる(パイプライン動作をしているので，書き換えを行ってからキャッシュに反映されて命令フェッチできるまでに数命令分の遅れがあるはずであるが)し，メモリもむだにならない．i486はハードウェアアーキテクチャこそRISCであるが，命令セットアーキテクチャは「バリバリの」CISCなので，メモリアクセスが非常に多い．命令キャッシュとデータキャッシュの同時アクセスによるパイプライン停止が頻繁に発生していると思われるのだが，どのように対応しているのだろう(詳細情報は公開されていないようだ).

Intelも，Pentium以降は命令キャッシュとデータキャッシュを分離した．命令キャッシュとデータキャッシュのアクセスの競合をなくすためだという．ただし，これまで動いていたプログラムが動かなくなっては互換性に問題が生じるので，命令書き換えは依然としてサポートしているようである．

なお，ハーバードアーキテクチャに対応して，命令とデータの経路が共通な方式を**プリンストンアーキテクチャ**ということもある(あまり一般的ではないが).これは，初期のコンピュータを提唱したフォン・ノイマン教授がプリンストン大学に属していたことに由来する．

● **1次キャッシュと2次キャッシュ**

図1で示したメモリ階層がMPUの内蔵キャッシュにも当てはまる．チップに内蔵できるキャッシュ(1次キャッシュ)の容量にはチップサイズから来る上限値がある(64K～128Kバイト程度)ので，少し低速で大容量(128K～4Mバイト程度)のSRAMをキャッシュ(2次キャッシュ)として外付けする構成が考えられる．この場合，外付けという性格上2次キャッシュは

物理アドレスキャッシュである．MIPSのR4000/R5000/R10000などは，この構成を採用している．また，2次キャッシュがチップに内蔵されるようになった最近では，外付けの3次キャッシュをサポートするMPUも登場してきている．

ところで，最近では，1次キャッシュ(Primary Cache)，2次キャッシュ(Secondary Cache)，3次キャッシュ(Tertiary Cache)は，それぞれ，L1キャッシュ(Level 1 Cache)，L2キャッシュ(Level 2 Cache)，L3キャッシュ(Level 3 Cache)と呼ばれる場合も多いようだ．

● プリフェッチ

プリフェッチとは特定の命令(プリフェッチ命令)を実行することで，パイプラインを止めることなく，キャッシュ(通常はデータキャッシュ)へのリフィルを強制的に行う．同時に，データキャッシュへのアクセスが発生するリード命令やライト命令を実行しない限りパイプラインを止める必要はない．ただし命令キャッシュは，基本的には，絶えずアクセスされているので，パイプラインを止めずに命令キャッシュへのプリフェッチを行うことは事実上不可能である．したがって，命令キャッシュへのプリフェッチ命令は，もし存在しても意味がない．

さて，どの領域をプリフェッチするかはプログラマ(やコンパイラ)が明示的に指定する必要がある．近い将来にアクセスする領域を指定しておけば，データキャッシュアクセスと競合しない限り，バスのアイドル期間を縫ってキャッシュへのリフィルが行われる．プリフェッチは有効に使えばかなり効果がありそうである．

プリフェッチが行われる契機はプリフェッチ命令によることが多い．しかし，最近ではハードウェアで自動的にプリフェッチを行う場合もある．キャッシュの無効な部分をそのままにしておくのはもったいないので，できるだけ有効データを取り込んでおこうという考え方である．ハードウェアプリフェッチを実装すれば，プリフェッチ命令を用いなくても，バスのアイドル時間を縫って自動的にプリフェッチすることが可能である．この機構はとくに命令キャッシュに対して有効である．上述したように，命令キャッシュは絶えずアクセスされるので，プリフェッチの契機となるアイドル時間は発生しにくい．命令キャッシュのプリフェッチを効率的に行うには，リードしながらライト可能な機構をキャッシュに埋め込む必要がある．

ただし，命令キャッシュへのプリフェッチは無条件に連続して行えばいいというものではない．実行する命令列には定期的に分岐命令が出現し，まったく別のアドレスに分岐する可能性もある．分岐命令の次までもどんどんプリフェッチするのは効率が悪い．そこで，命令フェッチ部分にプリデコード機能を設け，分岐命令と思われる命令コードに行き当たるとプリフェッチを停止する方式が採用される．あるいは，分岐予測機能もプリフェッチ機構に含め，分岐命令に行き当たっても，分岐予測をしながら，予測した分岐先からプリフェッチを継続する場合もある．この考えを推し進めていくと，Pentium4が採用している実行トレースキャッシュになる．

● フェッチバイパス

多くのMPUはメモリアクセスがキャッシュにヒットすることを前提に設計されている．ノンブロッキングキャッシュは別であるが，キャッシュミスが発生するとリフィルが完了するまでパイプラインが停止する．命令の連続実行という観点でいうと，一度止まってから最高速で動き，また止まってから最高速で動く……という動作を繰り返しているというイメージであろうか．

そこで，誰もが思いつくのが，止まっている時間がもったいないので，リフィルしているデータをキャッシュに書き込むと同時にMPUにも渡してしまうという方式である．そうするとリフィル中もパイプラインが動作できる．ただし，その間は，命令の実行スピードはバスクロック程度になってしまう．これが**フェッチバイパス**である．

パイプラインクロックとバスクロックに差がありすぎる場合は，バイパス効果はあまり期待できないが，差がほとんどない場合は非常に有効である．バスサイクルは常に起動されているわけではなく，バスサイクルとバスサイクルの間には数クロックのアイドル期間が生じる．リフィルする命令数が少ない場合は，この数クロックの間にそれらの命令を実行できてしまう．つまり，このような場合は，バスサイクルと同時に命令を実行するのも，命令をキャッシュに取り込んでから命令を実行するのも，ほとんど同じ実行効率となる．

MIPSでは，R3000の命令実行においてフェッチバイパス方式を採用しており，「命令ストリーミング」と呼んでいる．

● キャッシュロック

高速にアクセスできる作業領域をMPUのチップ内

に持ちたい場合がある．あるいは性能にクリティカルな命令領域を常に高速にアクセスしたい場合がある．このような機能を，キャッシュを用いて実現することができる(もちろん，そういう機能が用意されていれば)．キャッシュの特定のラインにキャッシュミスが生じても新たなリフィルによって更新しなければよい．この機能をキャッュロックという．

キャッシュロックの実装方法はさまざまである．現在のキャッシュの構成方式(nウェイセットアソシアティブが主流なので，それを念頭におく)は，タグ部の中にロックビットを設けてライン単位にキャッシュロックを指定する方法のほかに，特定のウェイをすべてロックしてしまうという方法もある．

キャッシュロックの目的は，ある一定量の領域をプログラムの作業領域として確保することである．細々とロックの指定をしなくても，せいぜい4～8ワードのライン単位に一つのウェイに属する1,000ワード(4Kバイト)以上の単位でロックできれば十分である．ダイレクトマップ方式のキャッシュでキャッシュロックを採用するMPUもあるが，さすがにこの場合はウェイ単位でロックを指定することは非現実であろう．もともとウェイが一つしかないので，キャッシュがまったく機能しなくなってしまう．

● キャッシュ可能領域

MPUによっては周辺デバイスにアクセスするためのI/O命令(IN，OUTなどの命令)が用意されているが，メモリ空間の一部をI/O領域として割り付けること(メモリマップトI/O)を前提とし，I/Oアクセスの専用命令がない場合も多い．I/Oデバイスは同じアドレス(I/Oポート)をリードしても同じ値が返ってくるとは限らない．つまり，I/O空間をキャッシュしてはならないのである．

このほかにも，フレームバッファやDMAの作業領域など，キャッシュ対象にされると具合の悪いメモリ空間もある．このように，同じメモリ空間の中でもキャッシュしてよい(メインメモリのコピーをもってよい)領域としてはいけない領域が存在する．

これをどのように区別するかが問題である．大抵のMPUでは，ある領域がキャッシュ可能であるか否かをMMUで仮想アドレスのページ単位に指定できるようになっている．仮想記憶をサポートしないMPUでは物理アドレスでキャッシュ可能空間とキャッシュ不可能空間が区別されている．MPUの専用端子でキャッシュ可能/不可能を指定する方式もある．つまり，キャッシュリフィルのバスサイクル中に，ある専用端子をアサート(アクティベート，活性化)することにより，その時点でデータバスに乗っているデータはキャッシュに入れないとする方式である．再び同じアドレスにアクセスする場合は，(キャッシュに入っていないので)キャッシュミスするため，もう一度リフィルが発生する．ここで再び専用端子をアサートすれば，そのデータもキャッシュに入ることなくMPUに渡される．このようなしくみでキャッシュ不可領域を実現できる．もっとも，ライトバック方式のキャッシュでは破綻をきたすかもしれないが．

● バススヌープ

DMAなどメインメモリに直接アクセスする処理を行う場合，メインメモリとキャッシュの内容が食い違うという現象が発生する．I/Oとは異なり，DMAによるデータは通常はキャッシュしてもかまわないデータであるが，食い違いをMPUに通知し，メインメモリとキャッシュの整合性を回復する必要がある．このための一手法がバススヌープである．バスモニタともいう．

具体的には，アドレスを指定してそのアドレスにヒットするキャッシュラインを無効化する．この場合，MPUの外部からキャッシュを無効化するアドレス(多くの場合，アドレスバスが使用される)を入力し，専用端子をアサートすることでスヌープが実現される．

図12にバススヌープ機能の概念図を示す．しかし，バススヌープ機能をもたないMPUも多い．DMAコントローラは転送の終了時にTC(Terminal Count)割り込みを発生するので，MPUはその割り込みを検知して割り込みを発生し，割り込みハンドラ内でDMAされたアドレスに対応するキャッシュラインを専用命令(キャッシュを内蔵するMPUにはたいてい用意されている)で無効化すれば事足りるからである．

このようにDMAの場合は割り込みによってソフトウェアで処理できるが，メインメモリを共有するマルチプロセッサ構成では，他のプロセッサがメインメモリの内容を書き換えたのを検知するのは容易ではない．この場合は，割り込みを使用すると処理が繁雑になるので，バススヌープが活用される(というか，バススヌープ機能がなくてはマルチプロセッサ対応とはいえない)．

● ウェイ予測

MPUの設計において，キーポイントの一つが消費電力の削減である．MPUの中でもっとも電力を消費

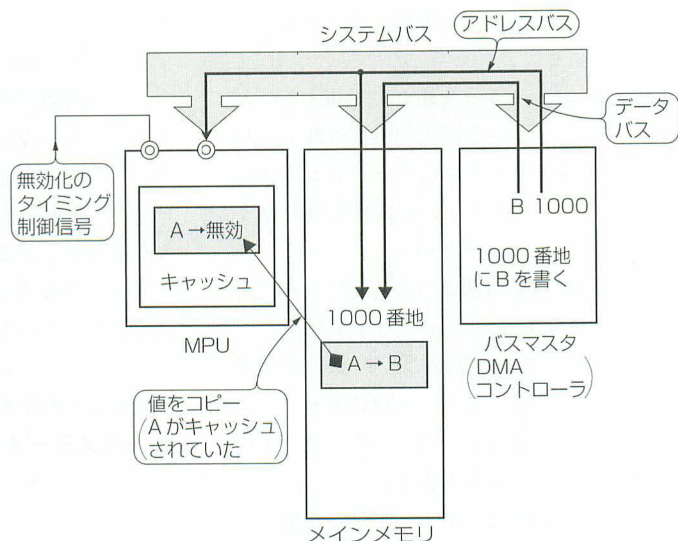


図12　バススヌープの概念

する部分は，じつはキャッシュであり，総消費電力の半分程度がキャッシュで消費されているといっても過言ではない．消費電力を削減するために，キャッシュの回路設計においてはメモリセルのブロック分割などの手法が採られることが多い．

そして，ウェイ予測もキャッシュの消費電力を削減するために考案された技術である．対象となるのはnウェイセットアソシアティブ構成のキャッシュである．通常の構成では，あるアドレスが与えられたときに，すべてのウェイのタグ部とデータ部の内容を内部バスに出力し，キャッシュヒットするウェイがあればそこのデータを選択する．

いま，一つのウェイから1回に出力されるデータが1ワード(32ビット)であるとしよう．このとき，4ウェイセットアソシアティブの場合は4ワードのデータが同時に内部バスに出力され，128ビット分の値が変化する．バスを構成する各ビット線を0から1，または1から0に変化させるためにはトランジスタによって目的の値になるようにビット線を駆動しなければならない．このときに電力を消費する．バス上の値が変化しなければ電力はほとんど消費されない．

さて，ウェイ予測とは，内部バスを同時に駆動するのではなく，予測したウェイから順番に駆動していく(キャッシュのヒット/ミスも順番に判定する)方式である．上の例でいえば，1回当たりのバス上の信号変化は32ビット分のみになり，単純計算で，消費電力は1/4になる．ただ，片方のウェイのヒット/ミスを判断してから他方のウェイをアクセスするため，キャッシュアクセスのタイミングは厳しくなるという欠点がある．

図13に，2ウェイセットアソシアティブ構成時にウェイ予測を行う場合のタイミングチャートを示す．図でX，Yはウェイのどちらかを表している．どちらのウェイから先にヒット/ミスの判定を行うか(これが予測)については，LRUビットの値から予測する，前回のキャッシュアクセスと同じウェイを見るなどの方法が考えられるが，決定版という方法はないようである．ウェイ予測が当たればヒット/ミスの判定にロスはないが，予測が外れればヒット/ミスの判定に1クロック程度のロスが生じる．この場合，たしかに性能は若干低下するが，性能と消費電力のどちらに重点を置くかで，ウェイ予測の採用/不採用が決まるであろう．

事実，最近のMPUではウェイ予測を採用することがけっこうあるようだ．スーパースカラ方式のMPUではデコードした命令を命令キュー(FIFO)に蓄えておき，そこから命令実行ユニットに命令を発行する．命令デコードと命令発行の間には時間差があるので，ウェイ予測ミス時のペナルティは命令キューで緩衝され見かけ上はゼロになる．

ウェイ予測に関しては，基本特許が多く出願されている．最近，ウェイ予測を公表するMPUが多いが，特許の利権関係はどうなっているのだろうかと他人事ながら心配してしまう．

ところで，キャッシュアクセスはプログラムの実行においてもっともクリティカル(時間がかかる)部分である．この部分にウェイ予測を導入するとロジックが

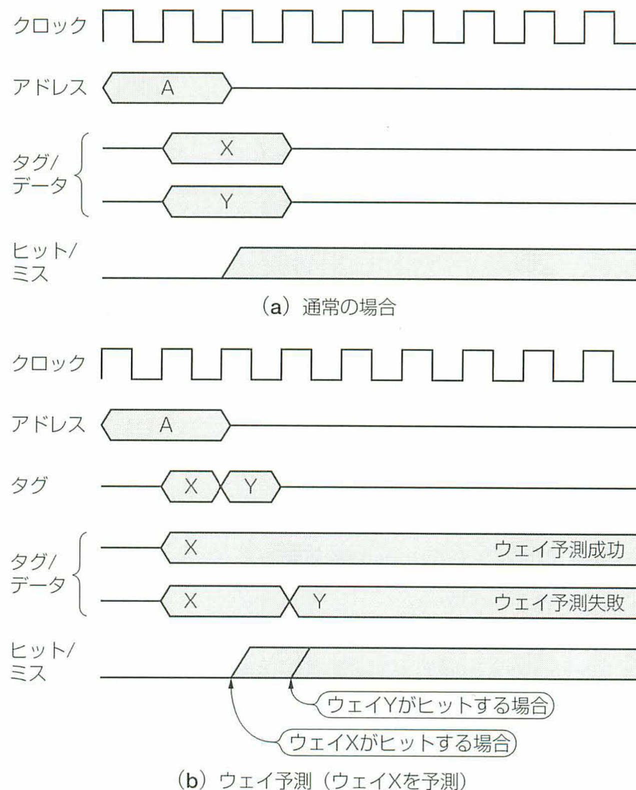


図13　ウェイ予測の概念

複雑になり，クリティカルパスが生じやすい．先に予測したタグを見てウェイのヒット/ミスを判断してから初めて別のウェイを参照することは，時間的(RAMのアクセスタイム)に厳しい．したがって，動作周波数を向上したい場合は，ウェイ予測は敬遠される傾向にある．

● 高性能MPUの命令キャッシュアクセス

最近の高性能MPUでは，命令キャッシュへのアクセスとデコード部分を実行部分と切り離して自律して動作させる．これを**デカップル**(decouple=分離)**方式**と呼ぶ．

つまり，命令実行パイプラインとは無関係に，命令を絶えずメモリから読み込み続けて命令キャッシュに格納している．この際，メモリから出てくるデータをプリデコード(おおまかなデコード，正確である必要はあまりない)して，分岐命令を探し当て，分岐予測機構と共同して次にアクセスするキャッシュラインを予測する．この機構を**図14**に示す．

デカップル方式では，デコード以降の命令実行パイプラインから見れば，欲しい命令は必ず命令キャッシュにヒットすることを期待している．これは，キャッシュを前提としたRISCでは当然の発想であるが，予測して命令を取り込み続けるフェッチ機構は複雑なので，高性能なMPUでしか採用されない．

デカップル方式で，デコード部分をフェッチ側に見るか，実行側に見るかは微妙なところがある．構造的にはデコード部はフェッチ側に近く，一般にデカップルと言えば，デコード部と実行部以降が命令キュー(リザベーションステーション)の前後で分離されていることを指す．

デカップル方式というか命令フェッチ機構の自律化は，今は亡き(?!)Alpha，MIPS R10000シリーズ，PowerPCが採用している．Pentium4に採用された実行トレースキャッシュも，似たような発想である．

● 仮想アドレスキャッシュは最近の流行か

動作周波数の高い最近のMPUでは，仮想アドレスキャッシュが流行になりつつある．キャッシュのアクセスが周波数向上のクリティカルパスとなることは稀ではないので，TLBを参照せずにヒット/ミスを決定できる仮想アドレスキャッシュは，キャッシュアクセスに余裕をもたせることができる．

最近では，ルネサステクノロジ(旧：日立製作所)/STマイクロのSH-5が全面的に仮想アドレスキャッシュを採用した．すでに記述したように，仮想アドレス

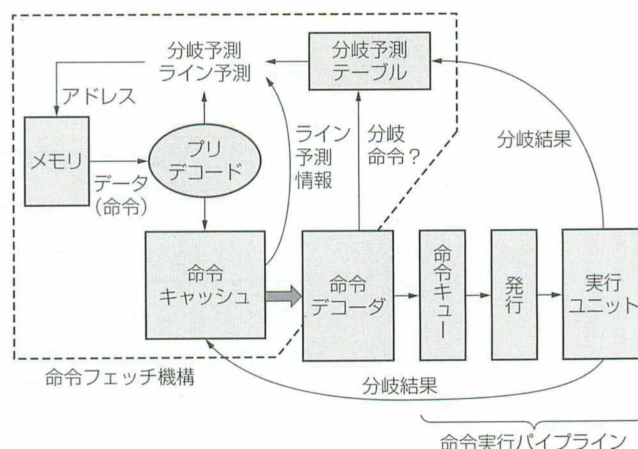


図14　命令フェッチの自律機構

キャッシュには，マルチタスク下において，同じ仮想アドレスで違う物理アドレスを指し示す場合がある(シノニムの問題)という欠点がある．

一般的にはプロセスID(タスクID)をキャッシュタグに付加することでシノニムの問題を解消する手法が採られるが，SH-5ではキャッシュミス時にTLBを参照し，その仮想アドレスに相当するTLBのエントリを無効化する手法を採る．このためのペナルティは5クロックという．この数値が大きいか小さいかはキャッシュミスの頻度によるが，仮想キャッシュが高速化のために有効ということになれば，今後もシノニム解決のためのいろいろな手法が生まれてくるであろう．

また，命令キャッシュに関しては，性質的にメモリ内容の変更を伴わないため，アクセスタイムが有利な仮想アドレスキャッシュが採用されるケースが増えているようである．つまり，ライトバックしないので，仮想アドレスに対応する物理アドレスが何であろうとあまり関係ない．キャッシュタグにプロセスIDを付加するか，タスク切り替え時に全エントリを無効化することで，ほとんどの場合は事足りる．

AMR10までのARMプロセッサやSH-5のほかにも，比較的新しいところでは，MIPSのRuby(R20K)が命令キャッシュに採用された．データキャッシュは，MIPSの従来方式である，仮想インデックス/物理タグである．

● ビクティム(Victim＝犠牲者)キャッシュ

1次キャッシュ，特にダイレクトマップ構成の1次キャッシュのヒット率を向上させるしくみとしてビクティムキャッシュがある．これは4～5エントリからなる小規模のフルアソシアティブキャッシュで，1次キャッシュからリプレースで追い出されたキャッシュラインを保持している．

1次キャッシュを参照する際，ビクティムキャッシュも同時に(あるいはビクティムキャッシュを優先的に)参照して，そこにヒットすればビクティムキャッシュからデータを供給する．ビクティムキャッシュのエントリは，基本的にLRU制御をされ，1次キャッシュから追い出されたキャッシュラインはビクティムキャッシュのもっとも参照されていないエントリに格納される．つまり，ビクティムキャッシュは追い出されたキャッシュラインのうちで最近参照された4～5ラインを保持することになる．これらのラインは，直前にリプレースされた1次キャッシュのラインがもっとも最近参照されたものだが，それ以外では1次キャッシュの他のラインよりも最近参照されたものである場合もある．

図15にビクティムキャッシュの構成を示す．一般に，キャッシュの参照はアドレスの一部をインデックスとして行うため，1次キャッシュの容量が少ないと，アドレスの競合が頻発して，同一のキャッシュエントリを追い出し合う傾向にある．このような競合を低減させるために，キャッシュのウェイ数を増加するという手段も採られるが，2ウェイ，4ウェイとするにつれ，キャッシュの容量(＝面積)が2倍，4倍となり，設計意図に反する場合がある．

ビクティムキャッシュを採用すれば,ある瞬間には,ダイレクトマップキャッシュの特定エントリが2～5ウェイになったように機能するので，少ない面積の増加でヒット率を向上させることができる．ある論文によれば，4Kバイトのダイレクトマップ方式のキャッシュに5エントリのビクティムキャッシュを付加したところ，アドレスの競合によるキャッシュミスが20

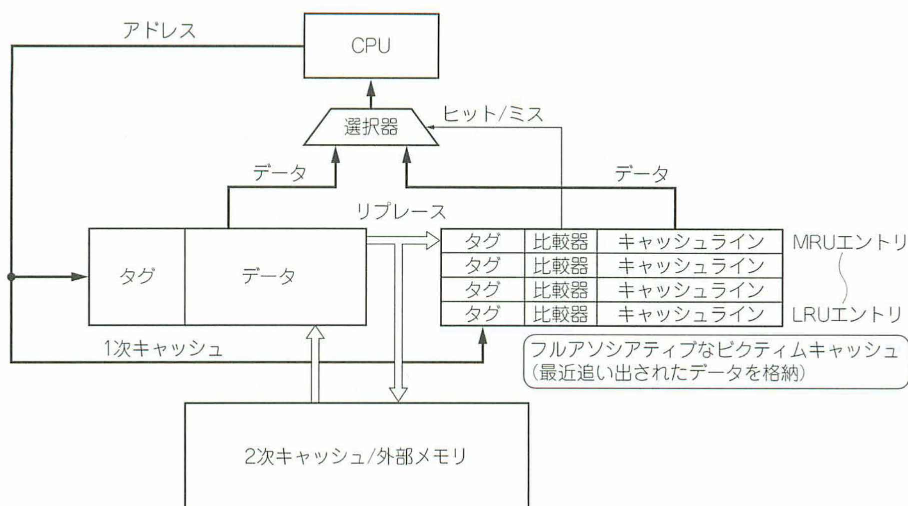


図15
ビクティムキャッシュの構成

～95％減少したという．

ビクティムキャッシュを実際に採用している製品としては，VIA TechnologiesのC3プロセッサがある．IntelやAMDがx86命令をRISCライクのμOPに変換して実行する方式を採用しているのに対して，C3はx86命令を直接実行するプロセッサとして有名である．CISCのままでも性能は出せるという主張をしているのだが，ベンチマークなどを見ると性能はお世辞にもいいとはいえない．

それはともかく，C3は64Kバイトの4ウェイ命令キャッシュと64Kバイトの4ウェイデータキャッシュを1次キャッシュとして内蔵し，さらに2次キャッシュとして64Kバイトの4ウェイのビクティムキャッシュを装備する．これを一般的な2次キャッシュと比較すると容量が小さいのが気になるが，小さな容量でもビクティムキャッシュとすること(メモリから直接リフィルを行うことはなく，1次キャッシュから追い出されたデータのみを保持する)で性能向上をねらったものであろう．

2003年1月22日にVIA TechnologiesはNehemiah(C5XL)コアを使ったC3シリーズのMPUを発表した．この新しいC3ではL2キャッシュであるビクティムキャッシュが，従来の4ウェイから16ウェイに変更になった．容量は64Kバイトと変更はない．やはり，ビクティムキャッシュはヒット率(＝連想方式)が性能に効くということなのだろう．

● イクスクルーシブ(排他的)キャッシュ

イクスクルーシブ(exclusive)キャッシュとはAMDがThunderbirdやDuron(Spitfire)以降に採用したキャッシュ構成である．

従来，1次キャッシュと2次キャッシュは階層構造をもっており，2次キャッシュの内容の一部分をキャッシュしたものが1次キャッシュであるという位置づけだった．つまり，1次キャッシュにミスした場合，2次キャッシュにヒットすれば，MPU外部のメモリからデータを読み込むより高速に1次キャッシュにデータをリフィルできる．データは，まず2次キャッシュにリフィルされ，その後，1次キャッシュにリフィルされる．この従来方式をAMDはインクルーシブ(inclusive)キャッシュと呼んでいる．

イクスクルーシブキャッシュは，1次キャッシュと(MPU内部の)2次キャッシュの内容を重複させない方式である．その動作原理は明らかにされてないが，キャッシュミス時に，1次キャッシュと2次キャッシュ(内部)の片方にしかリフィルしない．あるいは，いったん2次キャッシュにリフィルされてから1次キャッシュの内容と交換される．具体的には，次のような動作ではないかと想像される．

1) 1次キャッシュヒット
　そのラインが使用される．
2) 1次キャッシュミス＆2次キャッシュヒット
　2次キャッシュのヒットしたラインが1次キャッシュの置き換え対象ラインと入れ替えられる．そして，そのラインが使用される．
3) 1次キャッシュミス＆2次キャッシュミス
　2次キャッシュにリフィルされ，そのラインが1次キャッシュの置き換え対象ラインと入れ替えられる．そして，そのラインが使用される．

表1　x86系CPUのキャッシュ容量

CPUコード名	1次キャッシュ	2次キャッシュ	メーカ名
Thunderbird	128Kバイト	256Kバイト(内部)	AMD
Duron	128Kバイト	64Kバイト(内部)	
Prescott(Pentium4)	12Kμops+16Kバイト＝32Kバイト相当?	1Mバイト(内部)	Intel
Northwood(Pentium4)	12Kμops+8Kバイト＝24Kバイト相当?	512Kバイト(内部)	
Willamette(Pentium4)	12Kμops+8Kバイト＝24Kバイト相当?	256Kバイト(内部)	
Tualatin(Pentium Ⅲ)	32Kバイト	512Kバイト(内部)	
Coppermine(Pentium Ⅲ)	32Kバイト	256Kバイト(内部)	
Katmai(Pentium Ⅲ)	32Kバイト	512Kバイト(外部)	

格納された内容が重複しないので，キャッシュ容量が1次キャッシュと2次キャッシュを合わせた容量に等しくなる．インクルーシブキャッシュのキャッシュ容量は「(2次キャッシュの容量)±(1次キャッシュの容量)」で平均的には2次キャッシュ(内部)の容量に等しいというのがAMDの主張である．容量が増加した分，イクスクルーシブキャッシュでは，ヒット率が高いという計算である．

表1にx86系CPUのキャッシュ容量を示す．実際にAMDのMPUをみると，相対的な2次キャッシュの容量が少ない．しかし，キャッシュミス時にメモリからリフィルするためのペナルティを考えると，単に1次キャッシュの容量が増えただけのような気がしないでもない．

もともとAMDでは(Athlon以降)1次キャッシュの容量が非常に大きい．1次キャッシュだけでインテルでの2次キャッシュの半分の容量があるので，インテルとの相対性能を考えればこれで正解なのかもしれない．**表1**にIntelのMPUのキャッシュ容量も示しておくので参照してほしい．Intelの1次キャッシュの容量はゴミみたいなもので，通常2次キャッシュの容量でチップ仕様が語られることが多い．

ところで，イクスクルーシブキャッシュとビクティムキャッシュはどう違うのだろうか．おそらく同じものである．事実，VIA TechnologiesのC3プロセッサのデータシートでは，イクスクルーシブキャッシュとビクティムキャッシュを同一視している．もちろん，VIAのいうイクスクルーシブキャッシュとAMDのいうイクスクルーシブキャッシュが別物であるという可能性はあるが．

● 動作周波数とキャッシュのヒット率

ときどき，技術書を読むと「キャッシュ容量が少ないが，動作周波数が低いので，コスト/パフォーマンスを考えれば納得できる」という表現を見かける．一見すると，キャッシュ容量と動作周波数に関係があるのか不思議に思う．しかし，これは次のようなことを言っている．

MPUの動作周波数が高いほど性能が高い．これはある意味正しいが，キャッシュのヒット率(≒容量)を考えるとそうでもない場合がある．

たとえば，MIPS値が動作周波数の1.6倍のMPU-Aと1.0倍のMPU-Bを考える．ただし，MPU-AよりMPU-Bのほうがキャッシュの構造が優秀で，あるアプリケーションを実行したときのキャッシュヒット率は，MPU-Aが40％，MPU-Bが80％とする．ここで，MPUのバスクロックが133MHz固定で，それぞれのコアの動作周波数としてバスクロックの2倍(266MHz)と3倍(399MHz)の場合を考えてみよう．

この場合，それぞれの実質的なMIPS値は，フェッチバイパスを行う(キャッシュリフィル時の性能はバス速度に律速される)と仮定すれば次のようになる．

- MPU-A/266MHz動作時
  $(1-0.4)\times 133+1.6\times 266\times 0.4=250.04$MIPS
- MPU-B/266MHz動作時
  $(1-0.8)\times 133+1.0\times 266\times 0.8=239.40$MIPS
- MPU-A/399MHz動作時
  $(1-0.4)\times 133+1.6\times 399\times 0.4=335.16$MIPS
- MPU-B/399MHz動作時
  $(1-0.8)\times 133+1.0\times 399\times 0.8=345.80$MIPS

つまり，キャッシュミス時(1－キャッシュヒット率の部分)はバスクロックの速度(133MHz)で動作し，キャッシュヒット時はコアの周波数(266MHzまたは399MHz)で動作すると考える．

このとき，コアの動作周波数が266MHzの場合はMPU-Aのほうが性能が良い．しかし399MHzになるとMPU-Bのほうが性能が良い．この結果を見る限り，動作周波数だけでは性能を論じることができず，そこにはキャッシュのヒット率が大きく関与することがわかる．

この結果を一言でいうと，動作周波数が低いときは

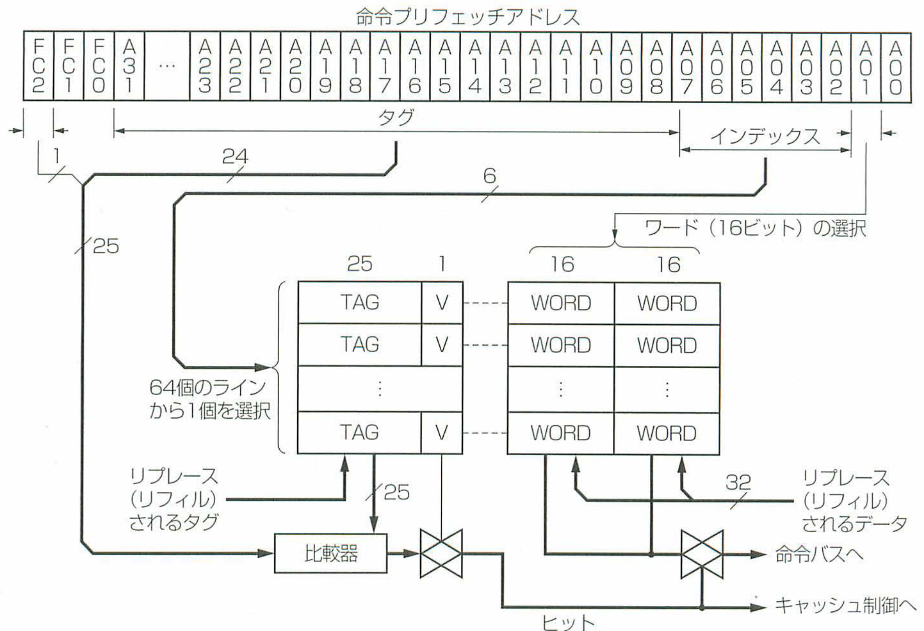


図16　MC68020の命令キャッシュ

命令の供給能力(≒バス速度)の影響を受けにくい．つまり，ヒット率が低いことにより発生するキャッシュリフィルの性能への影響が少ない．逆に動作周波数が高くなるほど，命令の供給能力が性能の足を引っ張るということである．

6 実際のプロセッサのキャッシュ構成

● MC680x0でのキャッシュ構成

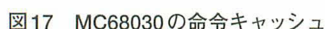
MC680x0シリーズではMC68020でキャッシュが内蔵された．ただし，命令キャッシュのみである．その構成を**図16**に示す．256バイトのダイレクトマップ方式で，1ラインは1ワード(4バイト)の容量をもつ物理アドレスキャッシュである．タグ部には機能コードのビット2(ユーザー/スーパバイザの表示)も含まれ，タグの比較時にアドレスと同時に比較される．MC680x0では同一の物理アドレスでも機能コードによって物理空間が区別されるからである．

MC68030では命令キャッシュに加えて，データキャッシュも内蔵された．**図17**にMC68030のデータキャッシュの構成を示す．256バイトのダイレクトマップ方式で，1ラインは4ワード(16バイト)の容量をもつ物理アドレスキャッシュである．書き込み制御はライトアロケート可能なライトスルー方式である．1ワードの容量がMC68020に比べ4倍に拡張されている．本来なら1ラインに1ビットあれば十分なバリッドビットがワードごとに用意され，全部で4ビットあるのが特徴である．キャッシュのリフィルを1ワード単位でも4ワード単位(バースト転送)でも行えるような設定が可能なためであろう．なお，命令キャッシュもまったく同じ構成をしている．**図18**にMC68030のキャッシュ制御レジスタ(CACR)を示す．この図を見ればわかるが，キャッシュロック(凍結)も可能である．

MC68040ではキャッシュ構成ががらりと変更された．**図19**にそのキャッシュの構成を示す．4Kバイトの4ウェイセットアソシアティブ方式で，1ラインは4ワードの容量をもつ仮想インデックス物理タグキャッシュである．書き込み制御はMMUでページ単位にライトスルー(ライトアロケートはしない)方式とライトバック方式を選択できる．リプレースメント方式はランダムである．

図20に命令キャッシュ，**図21**にデータキャッシュのライン構成を示す．バリッドビットはラインに1ビットのみとなった．不思議なのは**図20**でワード単位にダーティビットが用意されている点である．リフィルやライトバックはライン単位に行う(バリッドビットが1ビットしかないため)のでラインごとに1ビットあれば十分なはずなのだが．おそらく，ライトのバスサイクルを減らすために，真にダーティなワードのみをライトバックするためなのだろう．これにより，ライトバスサイクルの節約になる．

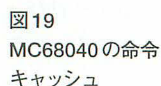
図22にMC68040のキャッシュ制御レジスタを示


図17 MC68030の命令キャッシュ

31 … 14	13	12	11	10	9	8	7 … 5	4	3	2	1	0
000000000000000000	WA	DBE	CD	CED	FD	ED	000	IBE	CI	CEI	FI	EI


WA：ライトアロケート
DBE：データバーストイネーブル
CD：データキャッシュのクリア
CED：データキャッシュの中のエントリのクリア
FD：データキャッシュの凍結
ED：データキャッシュのイネーブル
IBE：命令バーストイネーブル
CI：命令キャッシュのクリア
CEI：命令キャッシュの中のエントリのクリア
FI：命令キャッシュの凍結
EI：命令キャッシュのイネーブル

図18 MC68030のキャッシュ制御レジスタ（CACR）


図19 MC68040の命令キャッシュ

TAG	V	LW3	LW2	LW1	LW0

TAG：22ビットの物理アドレス情報
V：バリッドビット
LWn：32ビットのデータエントリ

図20　MC68040の命令キャッシュのライン構成

TAG	V	LW3	D3	LW2	D2	LW1	D1	LW0	D0

TAG：22ビットの物理アドレス情報
V：バリッドビット
LWn：32ビットのデータエントリ
Dn：LWnに対応するダーティビット
INVALID：not V
VALID(Clean)：V and(not D0)and(not D1)and(not D2)and(not D3)
DIRTY：V and(D0 or D1 or D2 or D3)

図21　MC68040のデータキャッシュのライン構成

31	30 … 16	15	14 … 0
DE	000000000000000	IE	000000000000000

DE：データキャッシュのイネーブル
IE：命令キャッシュのイネーブル

図22　MC68040のキャッシュ制御レジスタ

す．それぞれのキャッシュのイネーブル(許可)ビットしかなく，キャッシュロック機能はなくなったもようである．

● i486のキャッシュ構成

i486のキャッシュは，8Kバイトの容量をもつ4ウェイセットアソシアティブ構成の物理アドレスキャッシュである．**図23**にi486のデータキャッシュのブロック図を示す．データキャッシュの書き込み制御はライトスルーで，リプレースは疑似LRUで行う．また，ライトアロケートは行わない．すなわち，リードミスでのみキャッシュをリフィルし，ライトミスではキャッシュをリフィルしない．

i486のキャッシュにはバススヌープ機能がある．プロセッサバスにキャッシュラインインバリデーションが発生すると，アドレスバスが示すアドレスに一致するエントリを無効化する．

● R4000のキャッシュ構成

MIPS R4000のキャッシュは，8Kバイトの容量をもつダイレクトマップ構成の仮想アドレスインデックス，物理タグキャッシュである．**図24**にR4000のデータキャッシュのブロック図を示す．書き込み制御はライトバック方式で，リードミスまたはライトミスでキャッシュラインをリフィルする．キャッシュミス発生時，リフィルされるエントリのダーティビットが1なら，リフィル前に，古いキャッシュラインをメモリまたは2次キャッシュにライトバックを行う．

R4000にもキャッシュのスヌープ機構がある．アドレスを指定して無効化を行うインバリデートプロトコルと，ラインの内容を更新するアップデートプロトコル(これはR4000MC/R4400MCのみ)がある．

MIPS系のプロセッサはダイレクトマップ方式を採用していることが多い．2ウェイセットアソシアティブキャッシュはハイエンドのR5000やR10000でしか採用されていなかった．最近では2ウェイセットアソシアティブ方式のものが増えてきているが，4ウェイはまだ珍しい．最近ではRuby(R20K)が4ウェイセットアソシアティブを採用しているのみである．ただし，MIPS社が提供するIPコアであるJade(4Kc)やOpal(5Kc)は1ウェイ(ダイレクトマップ)から4ウェイまでの構成を選択できるようになっている．とはいえ4ウェイ構成では消費電力が多くなるので，ウェイ予測などを行って電力を削減する工夫をしないと，組み込み用途には向かない．

ちなみに，Rubyはウェイ予測を行っている．また，最新のIPコアである24K(Topaz)は性能重視で4ウェイ構成のみになった．MIPS R4000のキャッシュを4ウェイアソシアティブ方式にすると，MC68040のキャッシュ構造に近くなる．

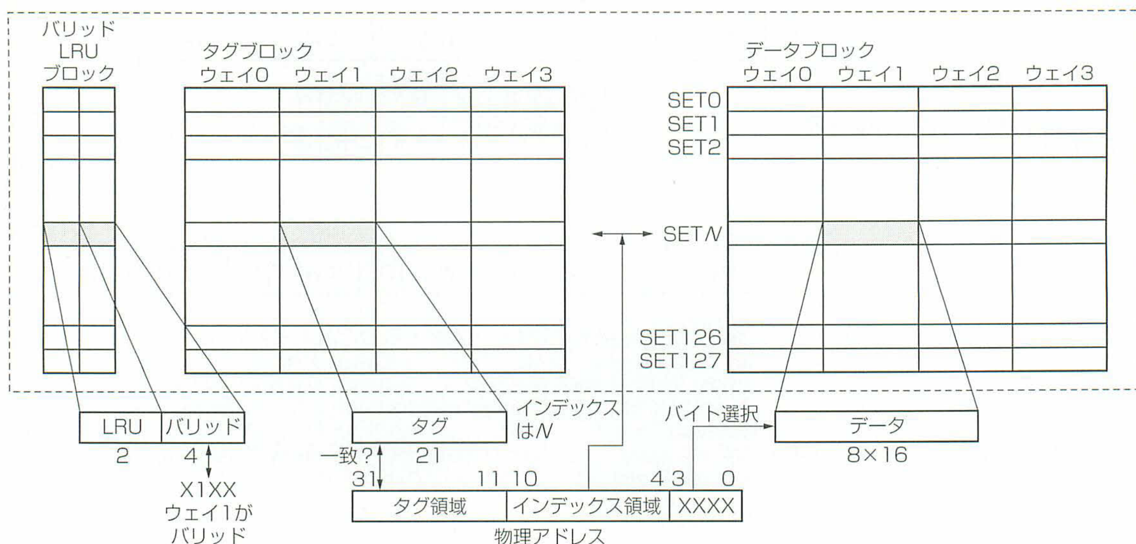


図23 i486のデータキャッシュ構成

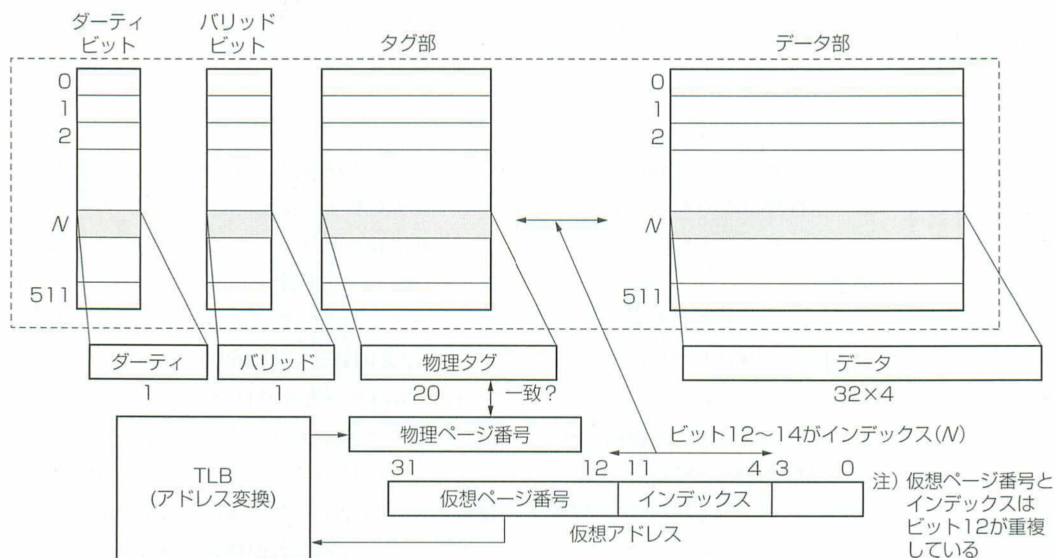


図24 R4000のデータキャッシュ構成(ページサイズ4Kバイトの場合)

まとめ

主として，MPUに内蔵されているキャッシュの概要を述べてきた．キャッシュの動作を少しでも理解していただければ幸いである．なお本章は，マルチプロセッサ構成時のキャッシュの動作については複雑になるので意図的に省いている．

ところで，本章では*n*ウェイセットアソシアティブにおける*n*個のダイレクトマップ形式のキャッシュを指すものとしてウェイという表現を使ってきたが，本来の意味が「*n*通りのセット」ということを考えると「セット」といったほうが正確かもしれない．まあ，ウェイと表現するのは筆者の職業病(?!)なので勘弁願いたい．また，ダイレクトマップという表現も正確にはダイレクトマップトである．

ところで，最近ではキャッシュのことをCASH(現金)との洒落で＄と記述することも多い(￥でないところが米国発祥の洒落であることを感じさせる)．たとえば，命令キャッシュやデータキャッシュは，それぞれ，I$，D$と略記されることもあるので覚えておこう．

Appendix 1

システムオンチップ時代のデバッグ手法

エミュレーション機能の基礎

ほとんどのMPUがもっていてマニュアルには通常記載されていない機能に，エミュレーション機能がある（デバッグ機能ともいう）．

ほかのアーキテクチャの命令コードを実行することもエミュレーションというが，ここではデバッガであるICE（In-Circuit Emulator）を実現する機能のことを指す．

● ICEとは

ICEとは，一言でいうとデバッガである．しかし，いわゆるソフトウェアのみで実現されているGDBのようなデバッガと違い，リアルタイム（実時間）エミュレーションが可能である．これは，実チップをターゲットシステムに実装してエミュレーションを行うことにより，実デバイスと同じAC特性やDC特性を実現したままのデバッグを可能にするものである．つまり，実チップと同じタイミングや環境でデバッグができる．

ICEは，実チップの代わりに専用プローブを実際のボードに差し込むことで，実チップの動作をエミュレートする（**図A**）．ユーザーからは，ICEというシステムが一つの実チップに見える．

その昔（今でも？），ICEは高価だったので，それを代替する手段もいくつか考えられている．たとえば，ロジックアナライザ（ロジアナ）をエミュレータの代わりにデバッガとして使う手法などである．ロジアナで取り込んだバスサイクルを逆アセンブルして，実行する命令列を表示するソフトウェアはけっこう活用されていた．しかし，この手法はICEにはかなわない．

● エミュレーション機能とは

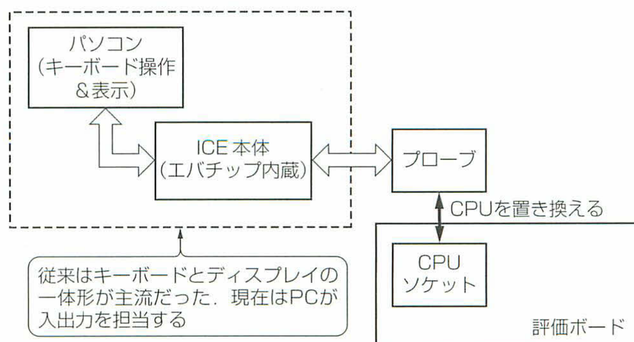
エミュレーション機能とは，ICEを実現するためにMPUが提供する機能のことである．デバッグ機能ともいう．具体的には，アドレスやデータの値によるトラップ機能（ハードウェアブレーク）や命令実行のトレース機能を指す．MMUを内蔵するMPUでは仮想アドレスを出力する機能もある．ハードウェアブレークとは，ブレークポイント命令をプログラムに埋め込んで，そこを通過した場合にブレークする，ソフトウェアブレークとは対称的にハードウェア自身が備えるブレーク機能のことである．

エミュレーション機能を実現するためには専用端子が必要なので，通常のチップ（通称本チップ）よりも端子数を増やしたエバリュエーションチップ（通称エバチップ）が製造される場合もあるが，基本的には本チップとエバチップは同一のダイ（チップ）であることが多い．つまり，外部端子に接続されていないだけで，ユーザーが手にする本チップもエミュレーション機能を内蔵している．しかし，その機能の使い方を知る方法はない．エバチップは，ICEメーカーに対して出荷される専用チップである．

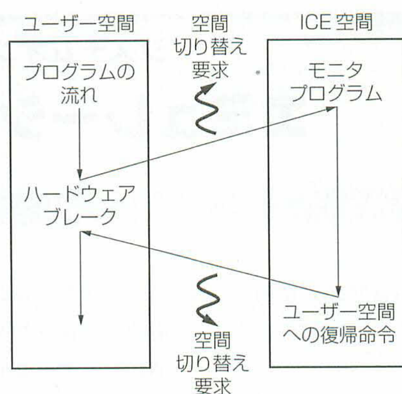
最近の流行は，エバチップを作らず，本チップにエミュレーション機能を内蔵させることである．この場合は，ユーザーがその機能を使おうと思えば使うことも可能である．ただし，エミュレーション機能の詳細はICEメーカー以外には公開されないのが普通なので，現実にユーザーが利用するのは難しい．

● フォアグラウンドモニタとバックグラウンドモニタ

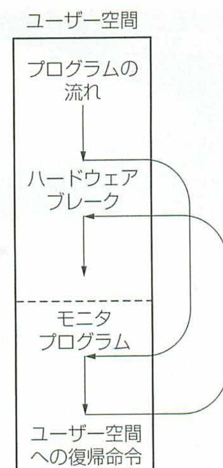
GDBなどの通常のソフトウェアデバッガとICEとが決定的に異なるのは，制御プログラムがユーザーの資源を占有するか否かである．ソフトウェアデバッガはユーザーのメモリ空間にロードされ，一つのタスクとして目的のプログラムをデバッグする．しかし，



図A　ICEの構成



(a) バックグラウンドモニタ



(b) フォアグラウンドモニタ

図B　バックグラウンドモニタ/フォアグラウンドモニタ

ICEは，制御プログラムのためにユーザーのメモリ空間を必要としない．これは，より現実に近い環境でプログラムをデバッグできるという利点のほかに，プラットホームの立ち上げ時，つまりボードにROMやRAMがまだ実装されていない段階でも，プログラムを実行することができるという利点がある．

ICEは二度美味しい．ボード設計の段階ではハードウェア屋が，ROMやRAM，あるいは外部I/Oなどに正常にアクセスできるかという，ボードのハードウェアのデバッグに使用できる．ROMやRAM上にプログラムが存在する必要はない．ボードが完成したあとは，ソフトウェア屋がソフトウェアをデバッグするのに使用できる．ソフトウェアのデバッグにおいては，ソフトウェアデバッガで十分という意見もあるかもしれない．しかし，ICEを使えば，ブレークポイント命令を埋め込むことのできないROM領域でブレークさせることもできるし，ある特定のアドレスに対してロードやストアを行った場合にブレークさせることもできる．あるいは，実時間で命令実行のトレースを行える．

さて，ICEの制御プログラム(モニタプログラム)はユーザー空間とは別の空間に置かれる．これは，ハードウェアブレーク発生時に，専用端子を活性化する．要するに，ハードウェア的にアドレス空間の切り替えを行うことで実現される〔図B(a)〕．あるいは，モニタプログラムを実行するための特殊な空間がMPUに内蔵されている場合もある．このような方式をバックグラウンドモニタと呼ぶ．簡易的なICEでは，モニタプログラムをユーザー空間に置く場合もある〔図B(b)〕．このような方式をフォアグラウンドモニタと呼ぶ．

● キャッシュ非内蔵時のエミュレーション機能

MPUの提供するエミュレーション機能は，MPUがキャッシュを内蔵しない場合と内蔵する場合で若干異なる．まずは，キャッシュを内蔵しないMPUが提供する，一世代前のエミュレーション機能について解説する．

(1) アドレストラップとデータトラップ

これは，MPUがアクセスするアドレスを指定してトラップを発生させる機能である．トラップが発生した後，制御はICEのモニタプログラムに移る．

キャッシュを内蔵しない場合，すべてのアクセスは外部バスに出力される．このため，バスサイクルを監視する機構を設けておけば，アドレストラップを実現するのは難しくない．目的のアドレスやデータが出力されたら，ICE空間に移行させるための強制ブレーク機能をMPUが備えていればよい．

より細かい条件でトラップを発生させるためには，現在起動されているバスサイクルの種類を表示するステータス出力があればよい．

(2) ステップ実行

1命令ずつ実行しながら，レジスタ内容やステータスを表示する機能である．この機能は，アドレストラップを次の命令のアドレスに設定することで実現する．

(3) 仮想アドレス出力

本チップには，端子数の制限から仮想アドレスを出力する機能はない．しかし，エバチップには，物理アドレスと同時に仮想アドレスを出力する機能がある．

専用のエバチップなしで仮想アドレスを出力するための手法としては，バスサイクルごとに仮想アドレスと物理アドレスを時分割で出力することが考えられ

る．しかし，本チップでは仮想アドレスの出力を行わないため，本チップとエバチップ(もしくはエミュレーションモード)でタイミングに差異が出るという欠点がある．

(4) トレース機能

トレース機能は，プログラムのデバッグには非常に有用な機能であるが，実現は難しくない．数十MHz以上で変化するMPUの状態をリアルタイムでユーザーが認識することは不可能なので，トレース機能はある程度までを実行した後に，そこまでの実行履歴を調べる場合に使用する．したがって，バスサイクルの出力をそのまま保存しておくことができれば，あとはソフトウェアでどうにでもなる．つまり，

- バスをトレースし，外部バッファに蓄える
- そのデータを逆アセンブルなどの加工を施して表示する

という手順で行うことができる．どの程度の命令数をトレースできるかは外部バッファの容量による．

(5) 分岐トレーストラップ

プログラムをデバッグするときに有用なのは，サブルーチンなど，命令の流れが不規則に変化する時点を認識することである．分岐の履歴を調べれば，プログラムの大まかな流れを知ることができる．

この機能を実現するために，分岐トレーストラップを提供するMPUもある．これは，分岐が発生した時点でトラップを発生させる機能である．ただ，分岐でいちいちトラップを発生させていると，プログラムの動作が実時間で動作させた場合と異なるのが欠点である．

(6) シーケンシャルブレーク

シーケンシャルブレークとは，その名のとおり，あるトラップ条件が成立した後に別のトラップ条件が成立するとき，初めてトラップさせる機能である．この機能は，トラップ発生時にICEの実行を止めるか否か，ICEのモニタプログラムで制御することで実現可能である．つまり，1回目のトラップが発生すると，そのことを記録しておき，そのままユーザープログラムに制御を戻す．そして，再びトラップが発生したときに，その旨をユーザーに表示すればよい．

ソフトウェア制御できるため，一見MPUが持つまでもない機能と思えるが，プログラムの動作を実時間で動作させた場合と一致させるためには必要な機能である．

● キャッシュ内蔵時のエミュレーション機能

キャッシュの内蔵は，MPUの性能に飛躍的な向上をもたらした．しかし，ICEにとっては嬉しいことではなかった．なぜなら，従来ICEが拠り処にしていたバスサイクルが発生しなくなったからである．

たまに発生するバスサイクルも，ほとんどがキャッシュリフィルのためのバスサイクルで，バスサイクルとそれを発生させた命令を1対1に関連付けることは難しい．このため，キャッシュ内蔵を当たり前とするRISCプロセッサ用のICEは，いつまで経っても実用的なものが登場しなかった．

当初RISCは，ワークステーションなどハイエンドの分野でしか使用されていなかった．この分野のデバッグは，昔ながらのロジアナで波形観測を行うのが普通であり，ICEの必要性を訴える人は少なかった．まさに職人芸の世界である．しかし，RISCが組み込み制御にも使われるようになるとICEの要望が高まってきた．組み込み分野では，従来からICEを使用してデバッグを行っていたので，「ICEのないMPUなんて使えない」という意見が多数派だったのだ．

このような事情もあって，キャッシュ内蔵のMPUでは，エミュレーション機能を実現するために新たな機能を内蔵することが必要になった．たとえば，MotorolaがDragonBallやColdFireシリーズに内蔵したBDM(Background Debug Mode)がそれである．BDMは，CPUのマイクロコードでデバッグ命令を実装し，専用のデバッグ端子を外部にもたせて，専用のケーブルでデバッガと交信するしくみになっている．

(1) ハードウェアブレーク

従来，ICEがバスサイクルを観測することで実現していたアドレストラップ，データトラップという機能をMPUに内蔵するようになった．従来も，アドレストラップ機能を内蔵し，ROM領域でもブレークポイントを設定できることを売りにするMPUはあったが，ロードやストア時のデータの値を指定してトラップさせる機能をもつものはほとんどなかった．

MPUは，ユーザーが使用するアドレストラップのほかに，ICE専用のアドレストラップやデータトラップ機能を提供するようになった．

トラップ発生時のICE専用空間への移行も，MPUがサポートする．

(2) トレース機能の実現

トレース機能も，従来はICEが外付けで実現していた機能をMPU内に取り込むことで理論上は可能である．しかし，それはMPU内部に巨大なトレースバッファを内蔵することを意味する．

ICE以外では使用しない機能で，チップの面積を増大させるのは好ましいことではない．そこで，最小限のハードウェアでトレース機能を実現する方式がいろいろ考案された．その基本原理は，分岐が発生したことを検出する点にある．従来一部のエバチップで採用されてきた分岐トレーストラップ機能はとくに有用である．

トレーストラップ機能に加え，分岐時に分岐先の仮想アドレスを出力できれば，ほぼ完全に命令実行をトレースできる．しかし，この仮想アドレスの出力というのが難問である．従来のエバチップでは，余分な端子数を追加することで実現していた機能である．たとえば本チップに，32本の仮想アドレス端子を追加することは不経済である．そこで考案されたのが，4本程度の専用端子を追加し，時分割で仮想アドレスを出力するという方式である．32ビットなら8回に分けて出力する．この方式の欠点は，分岐が連続して発生すると，仮想アドレスの出力が命令実行に追いつかなくなり，情報が失われることである．しかし，RISCの初期のICEでは，ないよりはまし，という割り切りで採用されていた．

もっとも最近のRISCでは，トレース出力が間に合わない場合は，ストール(一時停止)要求を外部回路から与えて待ち合わせをする機能があるようだが．

実際，トレース機能の実装はいろいろな制限から難しいことが多く，ハードウェアブレークだけでICEが作られることも多い．バックグラウンドモニタ機能を提供すれば，ハードウェアブレークしかできなくても，そこそこ使えるICEを構成できる．

● JTAGを利用したデバッガ

JTAGは，国際標準規格IEEE1149.1として普及している．JTAGは，機能の名称ではなく，この規格化作業を推進したグループの名称である．機能の名称は「バウンダリスキャン」である．IEEEの標準では，バウンダリスキャンアーキテクチャとそれにアクセスするためのシリアルポート(通称JTAGポート)が規格化されている．このJTAGポートは，本来，ボードのテスト用に考案されたものであるが，そのインターフェースをMPUのデバッグ機能に利用することが考えられている．

従来のICEでは，専用のデバッグ端子をMPUに装備する手法が採られていたが，現在はJTAGを利用する方法が主流である．JTAGは，もともとチップの回路テスト用に開発されたバウンダリスキャンのテスト手法である．JTAGの回路と外部テスト端子を利用し，デバッグ回路を追加することによりシステムのデバッグを行うのである．

JTAGを利用したデバッグ機能の拡張は，各社独自の仕様により行われている．こうしたオンチップデバッグ機能には，NECのN-WIRE，MIPS Technologies社のEJTAG(Enhanced JTAG)，Motorola社のCOP(Common On Chip Processor)などがある．本質はどれも似たようなものである．本来，N-WIREというのは，ICEメーカーであるHPが提唱したJTAGを使ったデバッグ機能であり，その意味ではNEC固有の規格ではない[注]．

ICEの短所は，ソフトウェアデバッガに比べて非常に高価な点である．従来品は，100万円を超えることも珍しくなかったが，JTAGを利用したデバッグ機能は，MPU内にデバッグ機能を内蔵し，外部とのインターフェースを最小限の端子本数(5～15本程度)に抑ることで安価(30～50万円)なICE構築を目指すものである．

JTAGと同時に用いられるオンチップのデバッグ機能は，従来と大差ないハードウェアブレークとトレース補助機能である．それを，JTAGという標準的なシリアルインターフェースでアクセスすることで，ICEのハードウェアの共通化を図ることができる．つまり，JTAGの先のデバッガ本体は，MPUの種類が変わっても同一のものを用いることができる(**図C**)．

JTAGのICEでは，従来のICEとは異なり，MPU自体がデバッグ機能を内蔵するので，エバチップは不要である．ボードテスト用のJTAG端子をデバッガに接続することでデバッグが可能になる．

● JTAGデバッガの実現例

JTAGデバッグ機能を有するMPUの一般的な構造を**図D**に示す．このように，MPUのデバッグユニットはバックグラウンドモニタ(ハードウェアブレーク

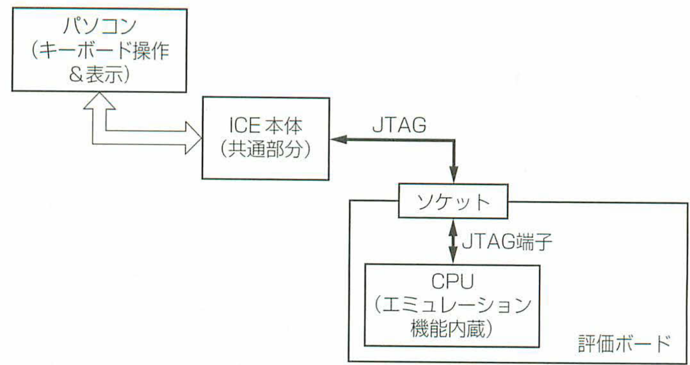
注：N-WIREは東芝のMIPS RISCであるR3900に初めて搭載された．東芝とYHP(横河・ヒューレット・パッカード)が開発した規格で，*N*本の信号線から構成されるので，そう命名された．その後，HPがJTAG仕様に変更し，さらなる拡張を行って現在の形になった．このとき，ハードウェアブレーク部とトレース部(N-Trace)が分離された．つまり，トレース機能のないものもN-WIREと呼ぶ．NECやARMも採用しているが，ARMはN-Traceと呼んでいる．また，日立のSH-3/SH-4に採用されたH-UDI(Hitachi User Debug Interface)も同様の形式であるらしい．

機能とモニタ機能）とトレース制御から構成される．

MPUが提供するハードウェアブレーク機能は直感的に理解しやすいが，トレース機能が実現される方法は興味深い．ここでは，JTAG ICE（N-WIRE）におけるトレース機能とモニタ機能の実現方法を簡単に説明する．

(1) トレースパケット

命令やデータのトレースは，トレースパケットと呼ばれる数種類の情報を，JTAGを通じて数ビットずつシリアル出力することで実現する．トレースパケットはMPU内部で生成される．トレースパケットの例を図Eに示す（実際のものとは異なる）．これは，パケットの種類を判別するTRCODEと，それに付随する情報で構成される．



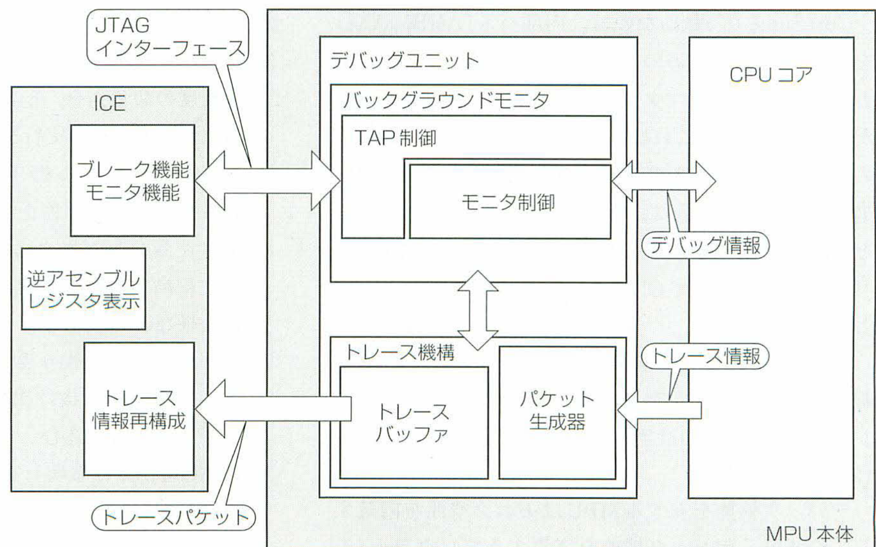
図C　JTAG ICEの構成

これらのトレースパケットは，アドレスの一致情報，分岐や例外の発生情報，その分岐先アドレスや分岐元アドレス，例外コードを示す．必然的に分岐に関する情報が多い．あとは，外付けのICEがトレースパケットを取り込んで，従来どおりのトレース表示を行う．

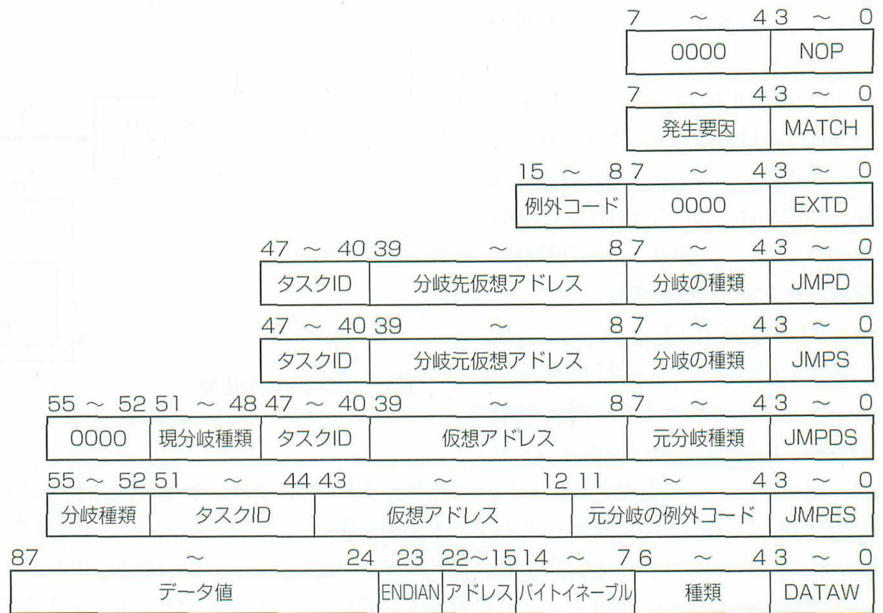
通常，ICEは低いクロックで動作している．反面，MPUは非常に高速なクロックで動作する．トレース機能は高速なMPUの状態をパケットにして出力するものであるから，ICEがそれを取り出して処理するためには，速度差を緩衝するバッファが必要である．まともなトレース機能を実現するためには，このトレースバッファのために多くの容量を必要とし，これがMPU内にトレース機能を実装する足枷となることが多い．このため，トレースバッファをMPUの外部メモリとしてサポートする場合（MIPSのEJTAGなど）もある．この場合は，MPUの動作クロックをあまり高速にできないのが欠点である．

MPUによっては，トレースパケットの出力が間に合わない場合は，外部からウェイト信号を入力して，出力を待ち合わせすることができるものもある．しかし，この場合はMPUの動作が停止するので，実環境と同じ実時間でのトレースにはならない．

ARM社では，トレースパケットの情報量を低減し，転送速度を向上させるためにETM（Embedded Trace Macrocell）と呼ぶ圧縮回路を提供する．ETMにはLarge，Medium，Smallといった種類があり，トレースのパケット量と速度に適した回路を選択することが



図D　JTAG ICEの動作



図E
トレースパケットの例

できる．トレースポートのビット幅も4ビット，8ビット，16ビットから選択できる．さらに，マルチコアSoC向けのトレース機能として，CoreSightテクノロジを提唱している．これは，従来のETMを改良して，マルチコア対応にしたものである．非同期(異なる周波数)で動作する各コアに専用のETMを備え，トレースファンネルというセレクタで非同期にパケットを排出する．

ETMにはいろいろなバージョンがあり，第3版のETMv3では最初のETMv1と比べると，命令トレース圧縮率は700%に，データトレース圧縮率を25%改善したとしている．

CoreSightで特徴的なのは，内部バス(AHB, AXI)をトレースするための規格を提供していることである．AHBトレースマクロセル(ATM)で，内部バス状況を監視できる．これは，ソフトウェアでの最適化用の意味が大きい(バスの使用効率を最大にする)．また，通常のETMと同様に，クロストリガ(ある値になるとデバッグ割り込みをCPUコアに通知する)を持ち，バスの状況によってCPUコアにブレークをかけることが可能である．

また，CoreSightは低価格用にシングルワイヤ(1本線)のデバッグを可能にする．実行制御用1本とトレース出力用1本の計2本でICEを構成できるらしい．

(2) モニタ機能

デバッグ機能を有するMPUはモニタ空間を内蔵する．つまり，デバッグ機能を実現するプログラム(モニタプログラム)はモニタ空間で実行される．しかし，4Kバイトや8Kバイト程度のモニタプログラムのためのデバッグ専用メモリを内蔵するのは不経済である．そこで考案されたのが，一つのモニタ機能を1～8命令程度で実行するものとし，その命令分の実行領域のみを内蔵する方式である．この領域をモニタ命令レジスタと呼ぶ．このレジスタは，1回に連続実行する命令の数だけ存在する(たとえば8本)．そして，具体的には，次のような手順でモニタ命令レジスタの内容を実行する．

1) アドレストラップやハードウェアブレークなどでモニタ空間に移行する
2) 命令実行が自動的に停止する
3) JTAGを経由して，モニタ機能を実現する1～8命令程度の命令列を，モニタ命令レジスタに書き込む
4) JTAG経由で命令実行を許可する
5) モニタ命令レジスタの内容が実行され，実行が終わると命令実行が停止する
6) モニタ命令レジスタの実行結果は一時的なレジスタに格納され，その値をJTAG経由で取り出し実行結果などの表示を行う
7) 3)～6)の処理を繰り返す
8) モニタ空間から抜け出すための命令をモニタ命令レジスタに書き込む
9) JTAG経由で命令実行を許可する
10) 制御はユーザープログラムに移る

第5章

仮想記憶/メモリ保護機能を実現するために

MMUの基礎と実際

ここではWindowsやLinuxなど，仮想記憶を使う場合に必須となるMMUについて解説する．通常は仮想記憶を使わないことの多い組み込み用途であっても，信頼性の高いシステムを構築するためにMMUのメモリ保護機能を使う場合もある．ここでは，アドレス変換，TLB(Translation Look-aside Buffer)，PTE(Page Table Entry)，メモリ保護機能について解説したあと，680x0系やx86系，MIPSやPowerPCのMMUについて解説する．

MMUとはMemory Management Unitの略語である．つまり，メモリ管理ユニットのことで，MPUの外部または内部にあって仮想記憶機能を実現する．単にC言語などでプログラミングするだけなら，仮想記憶の知識などはほとんど必要ない．しかし，プログラムサイズが一昔前に比べてはるかに巨大化しており，またマルチタスクが当然のように行われている昨今，その裏方にはMMUという「働き者」がいることを心に留めておいてほしい．

1 仮想記憶とは

● 仮想的に広大なメモリを用意する

その昔，まだメモリが高価だった頃，コンピュータに実装できるメモリ容量はわずかなものだった．時としてアプリケーションプログラムの容量は実際の物理メモリの容量を超え，そのようなプログラムを動作させるためにはアプリケーションプログラム側で細工する必要があった．

プログラムの性質として見ると，ある瞬間瞬間に実行されているのは全体の一部分にすぎない．そこで，プログラムをいくつかのブロックに分割し，必要な部分だけをメモリにロードして実行させ，不要になったらそのブロックを補助記憶装置(多くの場合，ハードディスク)に退避し，代わりにほかの必要なブロックを補助記憶装置から取り出して，新しいブロックと入れ替えるしくみが必要になる(**図1**)．しかし，実装されている物理メモリの容量を考慮しながらプログラミングをするのは効率的でないし，物理メモリの容量が変化すると，同じプログラムが使用できなくなってしまう．

そこで，このようなメモリ管理をOSに任せるしくみが考案された．これが仮想記憶の原点である．仮想記憶を利用すると，ユーザーは物理メモリを意識することなく，物理メモリの容量を超えるような巨大なプログラムを実行できる．

● マルチタスクを実現する

PCはもとより，現在では規模の大きな組み込み機器は，そのほとんどがマルチタスクで動作している．マルチタスクとは，複数のタスク(プログラム)を同時に物理メモリに置き，ある決められた順番に少しずつ(その多くは時分割で)実行していくものである．この場合，各タスクが必要とする全部の領域を物理メモリに割り当てようとすると，メモリに入りきらなくなってしまう．物理的に限られた容量しかないメモリを，多くのタスク間で分割して使用する手段が必要である．この場合も仮想記憶が有効である．そのため，仮想記憶といえば，現在ではマルチタスクを実現する手法として紹介されることが多い．

マルチタスクも，物理メモリを複数のブロックに分けて，そのブロックを各タスクに割り当てて実行させることで実現される．このような仮想記憶を行う場合，各タスクが自身に割り当てられた物理メモリのブロック以外をアクセスしないように保護する機能も必要になってくる．

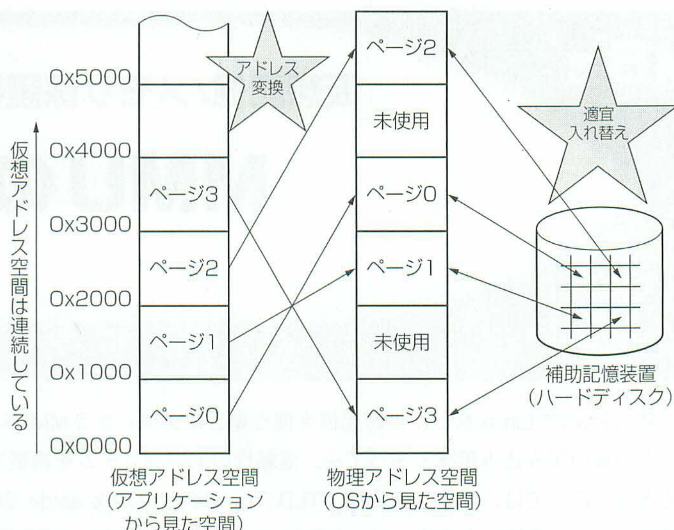


図1　仮想記憶のイメージ

● 現在ではページング方式が主流になっている

仮想記憶の方式としては，大きく分けて，**セグメント方式**と**ページング方式**がある．現在ではページング方式が主流なので，ここではページング方式を主体に話を進める．セグメント方式についてはコラム2で言及する．

ページング方式の場合，タスクのアドレス空間を分割したブロックを**ページ**と呼ぶ．また，不要になったページを補助記憶装置に退避したり，必要なページを補助記憶装置から復元する作業を**ページスワップ**と呼ぶ．

メモリのアクセス速度に比べてハードディスクのアクセス速度は非常に遅いので，ページスワップが頻繁に発生すると，プログラムの実行速度は低下する．しかし，プログラムとデータにはある程度局所性があるため，ページスワップがあまり発生しないことを期待して仮想記憶が実現されている．ところが，頻繁にページの範囲を超えて分岐が発生するプログラムや不連続な大量のデータを参照するプログラムでは，ページスワップの発生する確率が高くなる．このような場合は，そのタスクのページサイズを大きくすることで，ある程度ページスワップを回避できる．このため，MPUによってはタスクごとにページサイズを可変にできるようになっている．

2 アドレス変換

● アドレス変換とは？

PCにおけるプログラミングにおいて「このプログラムは物理アドレスの何番地に割り当てられるから」などと考えてプログラムを作る人は(OS屋などを除き)，まずいない．誰もが，自分の書いたプログラムは，たとえば「0番地から配置され無限の容量をもっている」と考える．つまり，プログラムはそれぞれ固有のアドレス空間をもっている．マルチタスクを行うということは，重複するアドレス空間をもつ複数のプログラム(タスク)を同時に物理メモリに割り当てて実行するということである．このような操作を可能にするためには，プログラムの中で想定されているアドレスを，実際の物理メモリに配置するためのアドレスに読み替えるしくみが必要になる．これが，**アドレス変換**である．

プログラムが想定しているアドレスは**仮想アドレス**(論理アドレスともいう)と呼ばれ，物理メモリに割り当てられるアドレスを**物理アドレス**(実アドレスともいう)と呼ぶ．アドレス変換とは，仮想アドレスを物理アドレスに変換する作業のことである．プログラムの仮想アドレス空間は，一定の容量をもつページに分割される．このページ単位に，仮想アドレスから物理アドレスの変換が行われる(**図1**)．

ページのサイズ(容量)はOSによってまちまちである．昔は1ページのサイズが2Kバイトのものが多かったが，現在は4Kバイトのものが多いようだ．4Kバイトは16進数で表現すれば1000バイトである．私見ではあるが，人間にとってなんとなくきりのいい数値なので，OS屋さんに好まれるのであろう．

それはともかく，仮想アドレスと物理アドレスの対応は，物理メモリ上に置かれたアドレス変換テーブルによる．この変換テーブルはページテーブルと呼ばれ，

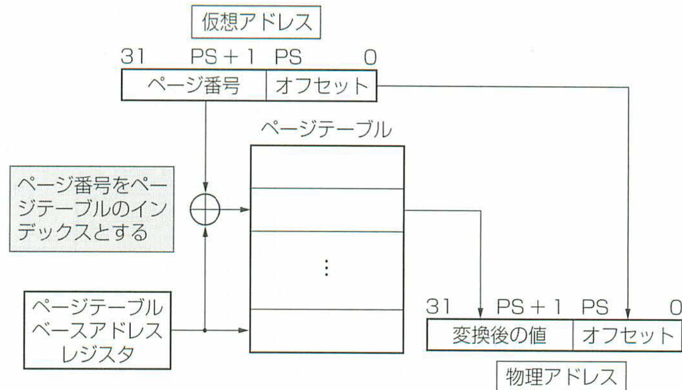


図2　1レベルのアドレス変換

通常4バイトまたは8バイト長のエントリの集まりである．これをとくにページテーブルエントリ（PTE）と呼ぶ．32ビットOSの場合，アドレスは32ビットで表現されるので，PTEには最低でも1ページあたり32ビット（4バイト）の領域が必要である．

もっとも，仮想アドレスと物理アドレスは，同一ページ内のオフセット（1ページが4Kバイトの場合はアドレスの下位12ビット）は一致するので，必要なビット数はもう少し少なくてよい．しかし実際には，そのページの保護情報のための情報やページスワップのための情報も必要になるし，ワード長（4バイト）またはダブルワード長（8バイト）のほうが（OSの）プログラムで扱いやすいので，一つの仮想アドレスに対して4バイトまたは8バイトのPTEが用いられるのが普通である．

● アドレス変換のレベル

さて，仮想アドレスが32ビット，1ページが4Kバイトの場合を考えよう．この場合，仮想アドレスの下位12ビットがページ内オフセット，上位20ビットがページ番号になる．このページ番号をインデックスとしてページテーブルを参照すれば，そのページに対応する物理アドレスを取り出すことができる．

なお，ページテーブルのベースアドレスはタスクごとに固有な値をもっていて，コンテキスト（タスクを性格づける情報）の一部である特権レジスタに格納されている．**図2**に仮想アドレスから物理アドレスを得る変換作業の概念図を示す．この図では20ビットのインデックスでページテーブルを参照するので，PTEの数は1M個必要である．PTEの容量は4バイトまたは8バイトなので，1タスクあたり4Mバイトまたは8Mバイトの物理メモリの容量がページテーブルのために必要になる．

しかし，タスクのもつアドレス空間は32ビット（4Gバイト）のすべての領域を使っているわけではなく，命令，データ，スタックなど，性質の異なる領域ごとにある程度塊になって存在している．このような場合，1M個のページテーブルエントリをすべて用意するのは不経済である．へたをしたら物理メモリがページテーブルだけであふれてしまうという状況も起こりかねない．そこで，ページテーブルを多段階に分けて参照する方法が考えられている．

この方式では，仮想アドレスのページ番号をさらにいくつかの領域に分ける．たとえば，20ビットのページ番号を上位12ビットと下位8ビットに分ける．この場合，上位12ビットをインデックスとして1段目のテーブルを参照し，2段目のテーブル（これがページテーブル）へのベースアドレスを獲得する．そして，下位8ビットをインデックスとして2段目のテーブルを参照し，物理アドレスを獲得する．この概念図を**図3**に示す．**図2**ではページテーブルを直接参照しているので1レベルのページング，**図3**では2回目でページテーブルを参照しているので2レベルのページングと呼ばれる．

最近のMPUでは，2レベルのページングでアドレス変換を行うことが主流だが，MC68030や68040では3レベルのページングを行うこともできる．

3　TLB

● TLBとは？

MPUが仮想記憶モードで動作している場合，仮想アドレスから物理アドレスへの変換を，いちいち物理メモリ上のページテーブルを参照しにいっていたのでは，その処理が命令実行のボトルネックになってしま

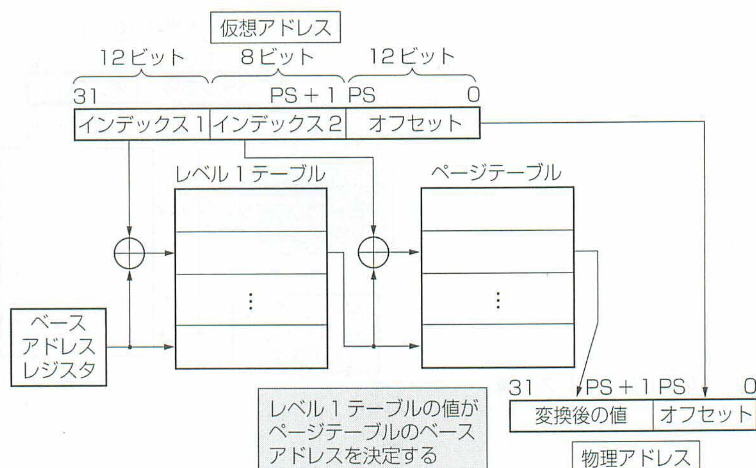


図3　2レベルのアドレス変換

う．それを避けるために，MPUは内部に**TLB**(Translation Look-aside Buffer)と呼ばれる変換テーブルをもっている．

日本語ではアドレス変換緩衝機構と訳されることが多い．モトローラは**ATC**(Address Translation Cache)，つまり，アドレス変換キャッシュと呼んでいる．その名のとおり，TLBとは，PTEをチップ内にキャッシュしたものである．

MPUはアドレス変換を行うとき，まずTLBを参照し，そこに目的の仮想アドレスと物理アドレスのペアが格納されていれば(TLBヒット)，その物理アドレスを用いて命令を処理する．もし該当する仮想アドレスがTLB内になければ(TLBミス)，物理メモリ上のページテーブルを参照しに行き，その値をTLBに登録する．また，TLBにはPTEと同様にメモリ保護などの情報が格納されており，TLB参照の際に不正アクセスがないかどうかチェックする．もし不正なアクセスである場合は，メモリ保護例外を発生する．以上がMMUの機能である．

ただし，最近のRISCチップでは，TLBを参照したとき，仮想アドレスが登録されていないと直ちに例外を発生して，TLBの内容を入れ替える処理をOSのプログラムに任せる．何度もメモリ上のテーブルを参照してTLBの内容を更新する処理は，実現が複雑であり，メモリアクセスはロード/ストア命令だけというRISCのポリシにも反する．何よりもパイプライン動作が妨げられてしまう．このためRISCでは，TLBの機能そのものがMMUの機能ということもできる．

● TLBの構造(連想方式)

TLBとは，仮想アドレスをタグとして内容を参照し，一致するタグがあれば対応するデータを物理アドレスとして出力する一種のキャッシュメモリである．その構造は参照の仕方により，次の3種類に分類できる．

- フルアソシアティブ方式
- ダイレクトマップ方式
- nウェイセットアソシアティブ方式($n \geqq 2$)

▶ フルアソシアティブ方式

フルアソシアティブ(Full Associative)方式は，TLBのエントリ数の数だけ異なる仮想アドレスを格納できる方式である．ほかの方式とは異なり，各エントリに格納される仮想アドレスに制限はない．通常は連想メモリという特殊なメモリで構成されるため，LRU処理(詳細は後述)が複雑になるため，多くのエントリをもたせることができない．現在の技術では50エントリ程度が限界と思われる．ただし，実装されているエントリをむだなく使用することができるので，少ないエントリ数でも高いヒット率(仮想アドレスを参照したとき，TLB内に存在する確率)を得ることができる．**図4**にフルアソシアティブ方式のTLBの構成を示す．

▶ ダイレクトマップ方式

ダイレクトマップ(Direct Mapped)方式は，もっとも単純な方式である．仮想アドレスが決まると，その仮想アドレスで参照するエントリが一意に決まってしまう．たとえば，256エントリのダイレクトマップ方式のTLBを参照する方法として，仮想アドレスのビット19〜12(8ビット)を使用してエントリをインデックスする方法が考えられる．これは，ページサイズが4Kバイトの場合である．仮想アドレスのビット31〜12がページ番号を表し，その下位8ビットである．

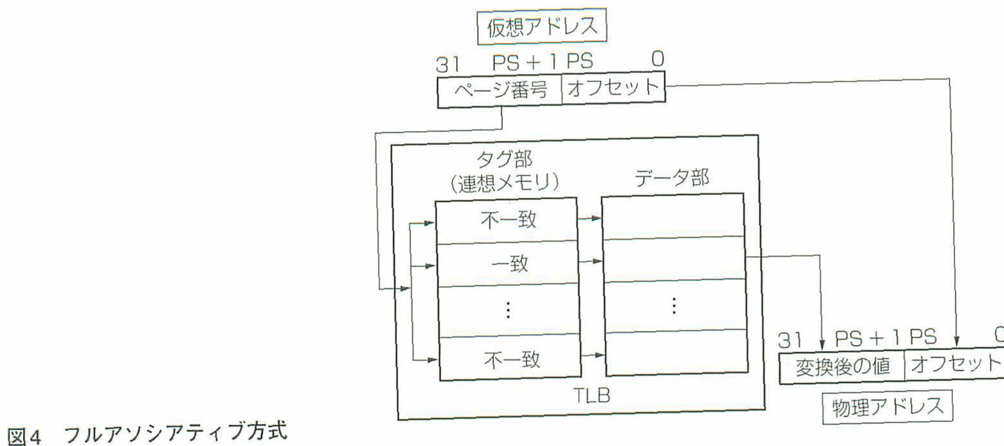


図4　フルアソシアティブ方式

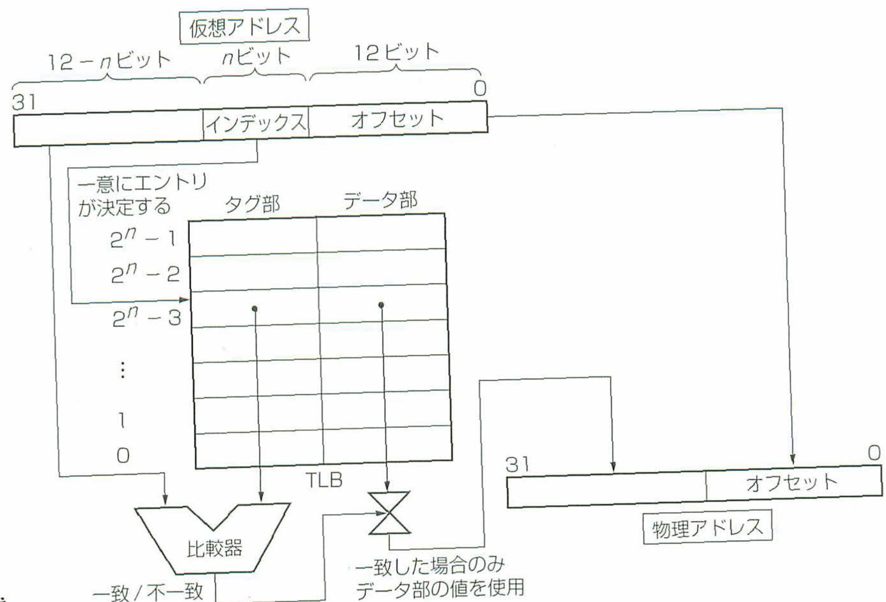


図5　ダイレクトマップ方式

8ビットのデータは256種類を識別できるので，仮想アドレスとTLBのエントリを1対1に対応させることができる．

ただし，この場合，下位8ビットが一致する仮想アドレスは異なるアドレスであっても同一のTLBエントリが参照されてしまう．プログラムの仮想アドレスが256通りでまんべんなく変化することは稀なので，場合によっては一度も参照されないエントリが存在する．逆に同じエントリが何度も参照され，前のデータを書き潰してしまうおそれもある．

ダイレクトマップ方式は，構造は単純でエントリ数を多くもたせることができるが，エントリ数を多くしないと高いヒット率は期待できない．**図5**にダイレクトマップ方式のTLBの構成を示す．

▶*n*ウェイセットアソシアティブ方式

***n*ウェイセットアソシアティブ**(n-way Set Associative)方式とは，ダイレクトマップ方式の改良版である．ダイレクトマップ方式のエントリをn系統用いて構成する．この方式も，LRU処理の制限からnの値は2または4であることが多い．簡単のために2ウェイセットアソシアティブ方式の場合で説明する．ダイレクトマップの場合と同様に，仮想アドレスが与えられるとエントリは一意に決定されるが，今の場合は2組のウェイ(エントリの集合)があるので，同時に二つのエントリに格納されている仮想アドレスと比較を行う．与えられた仮想アドレスがそのどちらかに一致していればヒットということになる．一般にウェイ数が増えるほどヒット率が向上する．**図6**に2ウェイセッ

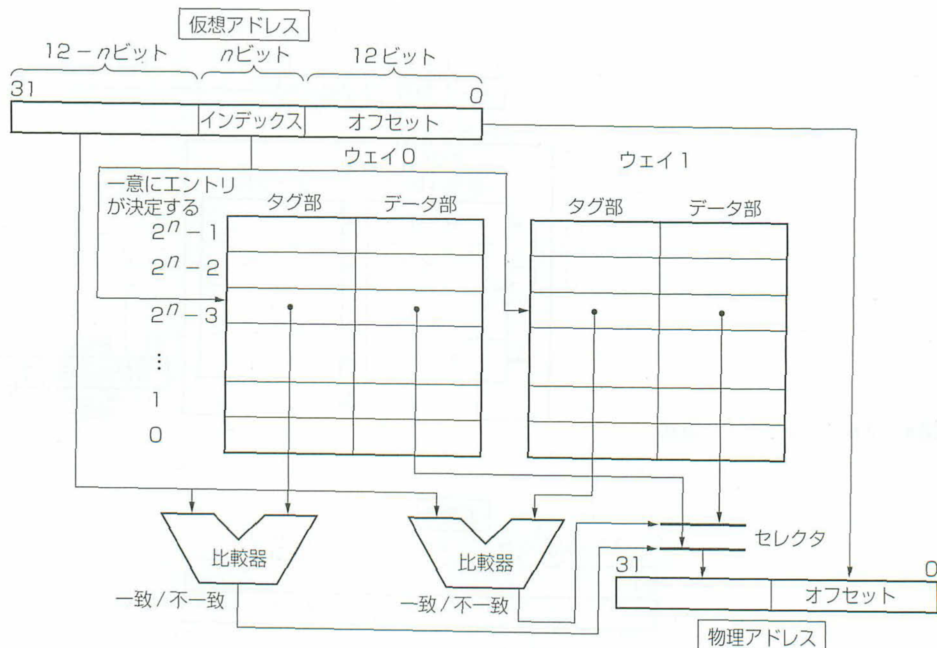


図6
2ウェイセットアソシアティブ方式

トアソシアティブ方式のTLBの構成を示す．

● TLBの更新方式

TLBのエントリ数には限りがある．エントリの中に有効なデータが入っていなければ，そこにアドレス変換の情報を格納していけばよいが，エントリがすでに有効なデータで占められていて，新たに変換の情報を登録できないことがある．この場合は，古い情報を追い出して新しい情報を書き込む(上書きする)ことになる．

追い出しの対象となるエントリを決定するためにもっとも多く使われるのが，LRU(Least Recently Used)という手法である．つまり，時間的にもっとも使用されていないエントリを追い出す．その実現方法は2ウェイセットアソシアティブ方式では簡単である．二つのエントリの組に対して1ビットのLRUビットを設ける．そのビットの値が0か1によって二つのうち対応するエントリをあらかじめ決めておく．そして，0側のウェイがヒットすればLRUを1側に，1側のウェイがヒットすればLRUを0側に更新する．もし，そのエントリに対応する仮想アドレスであって，どちらのエントリの内容とも一致しない仮想アドレスを変換しなければならない場合は，対応する物理アドレスを求め，LRUビットが示す側のウェイのエントリに上書きする．4ウェイセットアソシアティブの場合は四つのエントリに対して6ビットの情報でLRUを構成できる．フルアソシアティブ方式でのLRUはかなり複雑である．その方式が特許になるほどややこしいので，ここでは説明を省略する．

現実でも，エントリ数が多いTLBに対してはLRU方式を用いない．それでは，フルアソシアティブ方式の場合，追い出すエントリをどのように決定するのか．答は単純である．適当に決めるのである．具体的には(疑似)乱数を用いてエントリを決定する．これは，どのエントリの仮想アドレスも同じ程度に参照されていると仮定している．どのエントリが選ばれても恨みっこなしということである．

● タスク切り替えとTLB

タスクの仮想アドレス空間はタスクごとに固有である．意図的にほかのタスクのアドレス空間と一部の空間を共有させることもあるが，基本的には特定の特権レジスタの値で一意に規定される．この特権レジスタはコンテキストの一部であり，その値を基準として何回か間接参照を繰り返せば，最終的にページテーブルのベースアドレスを得ることができる．このため，タスクが切り替わればTLBの内容もそのタスクの仮想アドレス空間を反映したものに切り替わらなければならない．論理的にはタスクの数だけTLBが必要ということになる．しかし，現実的には，タスクの数の最大値を予測することは不可能であり，MMU内にいくつものTLBを実装するのはむだが多い(実質不可能)．

そこで，多くのMPUではタスクが切り替わるたびにTLBの内容を無効化してしまう．この方式では，

必要以上にTLBエントリを無効化してしまうおそれがあり，それがプログラムの実行速度の低下を招く．

たとえば，タスク番号0のタスクでは仮想アドレス1000番地しか参照せず，またタスク番号1のタスクでは仮想アドレス2000番地しか参照しない場合で，タスクが0→1→0と切り替わる場合を考える．このとき，最初は1000番地がTLBに登録されているが，タスクが1に切り替わる時点で無効化される．そして，タスクが再び0に切り替わるとき，1000番地はTLBに登録されていないので，再びメモリ上のアドレス変換テーブルを参照してTLBに1000番地を登録する必要がある．タスク1が1000番地を使用しないなら，このTLB入れ替え処理は余分である．しかし，他のタスクが使用する仮想アドレスを予測することはできないので，誤ったアドレス変換をしないように，古いタスクのアドレス変換情報は消去してしまわなければならない．必然的に，しなくてもよいTLB入れ替えが増加する．

その欠点を回避するために，TLBのタグ部にタスク番号を入れておき，タスク番号込みで仮想アドレスの一致を調べるという方式を採用するMPUもある．この方式だと，タスク番号1の仮想アドレス0番地と，タスク番号2の仮想アドレス0番地が同時にTLBに登録されていても(このような状況が発生するのはフルアソシアティブ方式のTLBに限られるが)，二つの0番地を区別することができる．タスクの切り替え時にTLBの内容を無効化する必要もない．TLB入れ替えは，本当に必要な場合にのみ行われる．

● TLBの分離

最近のMPUは，パイプライン処理で命令を実行している．命令フェッチやデータアクセスの前には仮想アドレスを物理アドレスに変換する必要があり，そのときTLBが参照される．何も考えずにMPUを設計すると，ある瞬間に，命令用のアドレス変換でのTLBの参照と，データ用のアドレス変換でのTLBの参照が同時に発生することになる．命令の仮想アドレスとデータの仮想アドレスは一般には一致しないので，二つの仮想アドレスで同時にTLBを参照することになるが，これは不可能である．どちらかの参照を遅れさせて，逐次的に参照をすることになる．

このときのパイプラインの乱れを嫌って，命令用とデータ用に二つのTLBを採用するMPUもある．キャッシュで命令とデータのデータパスをそれぞれ専用にもたせる構造をハーバードアーキテクチャと呼ぶが，そのTLB版と考えればよいだろう．実際，古くからハーバードアーキテクチャを提唱していたのはMotorolaであり，MC68040などは，命令とデータの2系統のTLBをサポートしている．

● マイクロTLB

命令とデータで同じ規模のTLBを用意するのは大げさだし，あまり効果はないように思える．なぜなら，データはともかく，命令のアドレスはシーケンシャルに実行され，分岐で初めて別のアドレスに切り替わるからである．分岐自身もページサイズの範囲を超えることは稀なので，命令のための仮想アドレスを変換しなければならない場合は(データに比べると)極端に少ない．そこで，命令用のTLBとして1～4エントリ程度の特別なTLBを採用するMPUもある．そのようなTLBは**マイクロTLB**と呼ばれる．

多くの場合，マイクロTLBは本体のTLBの内容をキャッシュしたもので，ページサイズも固定である．命令がマイクロTLBにミスした場合は，まず，本体のTLBを参照し，そこにヒットすれば，そこから物理アドレス情報をもってきて内容を更新する．TLBのページサイズがマイクロTLBのページサイズよりも大きい場合は，マイクロTLBでミスしても本体のTLBでヒットする確率が高いので，アドレス変換テーブル検索のためのメモリアクセスが発生することは稀である．また，このような構成であれば，メモリアクセスを発生させてTLBを更新するロジックが1系統分で済む．一方，命令TLBとデータTLBに分離されている場合は，それぞれ独立なTLB更新ロジックが必要である．

さらに，マイクロTLBの参照は，本体の巨大なTLBを参照するよりも少ない電力で行えるので，命令だけでなく，データに対してもマイクロTLBが採用されることもある．

4 PTE(Page Table Entry)の実例

● PTEとは？

TLBミス時，アドレス変換におけるメモリ内の変換テーブルのサーチは2～3段階のレベルに分けて行われる場合もあるが，最終的には，ページテーブルと呼ばれる**PTE**(Page Table Entry)が順次格納されているテーブルに突き当たる．PTEとは通常31ビット長のデータで，物理アドレス(オフセット部分を除く)と保護情報を含んでいる．PTEはページディスクリ

31–12	11	10	9	8	7	6–5	4	3	2	1–0
物理ページ番号	UR	G	U1	U0	S	CM	M	U	W	PDT

CM：キャッシュモード
G：グローバル
M：モディファイ（変更）
PDT：ページディスクリプタタイプ（存在，間接）
S：スーパバイザ保護
U：使用（参照）
U1：ユーザーページ属性1
U0：ユーザーページ属性0
UR：ユーザー用
W：ライト保護

(a) MC680x0のPTE（ページディスクリプタ）4Kバイトページ用

31–12	11–9	8–7	6	5	4	3	2	1	0
物理ページ番号	OR	IR	D	A	PCD	PWT	U/S	R/W	P

OR：OS用
IR：インテル予約
D：ダーティ（変更）
A：アクセス（参照）
P：プレゼント（存在）
PCD：ページキャッシュ禁止
PWT：ページライトスルー
U/S：ユーザー/スーパバイザ（保護）
R/W：リード/ライト（ライト保護）

(b) x86PTE

図7　PTEの実例

プタと呼ばれることもある．

ページテーブルに並んだPTEの意味は，先頭が仮想アドレス0(ページ0)に対応する情報，その次が仮想アドレス0x1000(ページ1，ページサイズが4Kバイトの場合)に対応する情報，その次が仮想アドレス0x2000(ページ2)に対応する情報，という具合になっている．何番目のPTEが使用されるかは仮想アドレスの値によって一意に決定される．**図7**にMC680x0で使用されるPTEとx86で使用されるPTEの実例を示す．PTEの情報のうち，物理アドレスに関しては説明不要と思うが，他のビットについて説明しておこう．

● MC680x0の場合

CM：キャッシュモード

このビットは対応する仮想ページの，キャッシュの可/不可，ライト制御(ライトスルー/ライトバック)，キャッシュ不可時にアクセスの逐次性を保証するか否かを示す．

G：グローバル

このビットはPFLUSH命令で使用する．PFLUSHはTLBのエントリを無効化する命令であるが，Gビットがセットされているページは無効化されない．

M：モディファイ

ライトアクセスで発生したTLBミスに起因する変換テーブルのサーチが行われた後，対応するPTEのMビットが自動的にセットされる．つまり，対応する仮想ページの内容が変更されたことを示す．

PDT：ページディスクリプタタイプ

ページディスクリプタ(PTEを含む変換テーブルエントリ)の種類を示す．それは，有効/無効，対応するテーブルまたはページがメモリ内に存在/不在，間接(中間)のディスクリプタか否かという情報を示す．間接ディスクリプタの場合は，次のレベルの変換テーブルの先頭を示す物理アドレスが格納されている．直接(最終)ディスクリプタの場合は，それがPTEであることを示す．なおMC680x0では，PTEに対応する物理ページの内容がメモリに存在することをレジデントと呼ぶようである．

S：スーパバイザ保護

スーパバイザモードのみで参照できるページであることを示す．Sビットがセットされていない場合は，スーパバイザモードでもユーザーモードでも参照できる．

U：使用

変換テーブルのサーチが行われた後，対応するPTEのUビットが自動的にセットされる．Mビットとは異なり，リード，ライト両方のアクセスでセットされる．対応する仮想ページの内容が参照されたことを示す．

U0，U1：ユーザーページ属性

MPUの実行に影響は与えない．それぞれの値が，UPA0，UPA1という端子状態に反映される(MC68040以降)．

UR：ユーザー使用

ユーザー(OS)が自由に使用してよいビット．MPUにとっては意味がない．

W：ライト保護

このビットがセットされている仮想ページに対し，ライトアクセスを行おうとすると例外が発生する．

● x86の場合

OR：OS用

OSが自由に使用してよいビット．MPUにとっては意味がない．

IR：インテル予約

将来の拡張用にインテル(メーカー)によって予約されているビット．現状，MPUにとっては意味がない．

D：ダーティ

ライトアクセスで発生したTLBミスに起因する変換テーブルのサーチが行われた後，対応するPTEのMビットが自動的にセットされる．つまり，対応する仮想ページの内容が変更されたことを示す．

A：アクセス

変換テーブルのサーチが行われた後，対応するPTEのAビットが自動的にセットされる．Dビットとは異なり，リード，ライト両方のアクセスでセットされる．対応する仮想ページの内容が参照されたことを示す．

PCD：ページキャッシュ禁止

このビットがセットされているPTEに対応する仮想ページはキャッシュアクセスを行わない．

PWT：ページライトスルー

このビットがセットされていると外部キャッシュ(L2キャッシュ)をライトバック制御にする．

U/S：ユーザー/スーパバイザ

このビットがセットされていないと，特権レベル3(ユーザーモード)では対応する仮想ページをアクセスできない．

R/W：リード/ライト

このビットがセットされていないと，特権レベル3(ユーザーモード)では対応する仮想ページに対してライトアクセスできない．

P：プレゼント

このビットがセットされていれば，対応する物理ページの内容がメモリに存在することを示す．このビットがセットされていない場合，ページフォールト(例外)が発生する．

これまでの説明を見ればわかるが，PTE内の情報は，どのMPUでも似たり寄ったりである．似ているのは，物理アドレス，アクセスがあったこと示す情報，ライトがあったことを示す情報，存在を示す情報，保護情報などである．個人的にはx86での名称がしっくりくるので，そちらを使って，以下に，OSがそれらの情報をどう利用するかを説明する．

● ページフォールト時の処理

ページフォールトとはPTEにアクセスした場合に，そのPTEが無効だったり，対応するページの内容がメモリに存在しないときに発生する例外である．

ページフォールトの処理には，Pビット(PDT = 00)を利用する．Pビットが0ならば，仮想アドレスに対応するプログラムの内容がメモリ内に存在しないことを意味する．初期状態ではPTE内の物理アドレス情報は決定されていない．ページフォールトが発生すると，OSはメモリ内に空いている領域を見つけ，そこに新しいページの内容を補助記憶装置(ハードディスク)からロードする(これをページインという)．このとき，見つかった空き領域の物理アドレスがPTE内の物理アドレス情報となる．メモリがすべて他のページに占有されていて空き領域がない場合は，どこかのページを補助記憶装置に追い出して(これをページアウトという)そこを使用する．ページインとページアウトの操作を総称してページスワップ(交換)と呼ぶ．

● ページスワップ時の疑似LRU(Least Recently Used)制御

ページアウトを行う場合，もっとも使用頻度の低いページを追い出すのが効率的である．Aビット(Uビット)を利用して疑似LRU処理を行い，使用頻度の低いページを決定する．そのために，OSは定期的に全PTEのAビットの状態をチェックする．そしてチェックが終わったら，ソフトウェアでAビットを強制的に0にクリアする．同時にそのPTEの内容がキャッシュされているTLBのエントリを無効化しておく．TLBにヒットする限り，PTEのアクセスが発生しないからである．こうしておけば，その後，同じ仮想アドレスに対するアクセスが発生すると再びAビットが1にセットされる．このような環境でAビットが1になる頻度を計数しておき，それがもっとも小さいページがアクセスのもっとも少ないページということになる．

なお，OSによっては，疑似LRU処理を行わず，単純なFIFO処理でページアウトするページを決定するものもある．これは，いちばん昔に変換したページから追い出していくというものである．あるいは，どのページを選択しても大差ないとして，ランダム処理で適当に追い出す候補を決める場合もあるかもしれない．

● ページスワップ時の補助記憶装置への無意味な書き戻しを制御

メモリ内に存在するページでも，そこに対して書き込みが行われていなければ，その内容は補助記憶装置に存在するもの(ページイン直前のもの)と同じである．つまり，そのページがページアウトの対象になっても補助記憶装置に書き戻す必要はない．補助記憶装置から読み込んだ新しいページの内容でメモリを書き潰してよい．これは処理時間の短縮につながる．この書き込み制御にはDビット(Mビット)を利用する．Dビットが0なら，そのページへの書き込みが行われていないことを示す．

● メモリマップトI/Oの実現

アドレス変換により，I/Oポートを仮想アドレスに対応させることもできる．その仮想アドレスをリード/ライトすることで，I/Oポートへのリード/ライトとみなすしくみを**メモリマップトI/O**という．これを実現する場合，そのページは非キャッシュ領域でなくてはならない．なぜなら，同じI/Oポートをリードしても同じ値が返ってくるとは限らないので，それがキャッシングされると都合が悪いからである．

PCDビット(CMビット)を使用すれば，そのページのキャッシングを禁止できるので，メモリマップトI/Oが実現できる．もっとも，PCDビットはI/Oポートでなく，フレームバッファなど，キャッシングされると都合の悪い領域の指定にも利用する．

また，I/Oポートはリード/ライトする順番が異なると意味が変わるので，アクセスの逐次化(プログラムで書いた順番にアクセスすること)を実現することも必要である．通常のMPUでは非キャッシュ領域に対するアクセスの逐次化は保証されているが，例外もある．それはMC680x0で，逐次化を明示的に指定する必要がある．

● 結局，キャッシュと同じ

アドレス変換で利用されるTLBはメモリに対する(ライトバック)キャッシュと同じように作用する．実際，OSの助けを借りるが，Aビットによる LRU処理，Dビットによるページアウト(ライトバック)処理は，キャッシュのそれと容易に対比できる．

Column 1 アドレス変換の効率化

アドレス変換で利用されるTLBはメモリに対する(ライトバック)キャッシュと同じように作用する．実際，OSの助けを借りるが，AビットによるLRU処理，Dビットによるページアウト(ライトバック)処理は，キャッシュのそれと容易に退避できる．ただし，通常のキャッシュミスのリフィル処理はあまり時間がかからないが，TLBミスによる入れ替え処理は，メモリアクセスが多いため，その10倍程度の時間がかかる．ページフォールトが発生するような場合は，100倍程度の時間は優にかかってしまう．アドレス変換を効率的に行うためにはTLBミスの確率を減少させる(ヒット率を上げる)ことが肝要である．

キャッシュのヒット率を向上させる手法と対比してTLBのヒット率を向上させる方法を考える．キャッシュのヒット率に関しては第4章を参照のこと．

● 容量を増やす

これはTLBのエントリ数を増やすことに相当する．通常，TLBはフルアソシアティブで参照されるが，エントリ数が増えると，回路規模の問題で4ウェイ，あるいは8ウェイセットアソシアティブと簡略なものにならざるをえない．同じ容量(エントリ数)でのフルアソシアティブとセットアソシアティブを比較すると，フルアソシアティブのほうがヒット率は高い．つまり，エントリ数を増やしても，セットアソシアティブにすることでヒット率が上がるとはかぎらない．

この辺のトレードオフは回路規模と欲しいヒット率を考慮してアーキテクチャを決定する必要がある．とはいえ，一つのタスクでアクセスされる仮想アドレスの数はせいぜい20程度だと思うので，TLBの構造が性能に大きな影響を与えるとは思えないのだが….

● ラインサイズを増やす

これはページサイズを大きくすることに相当する．TLBのヒット率が向上する反面，ページアウト時の処理時間が長くなってしまう．それを防ぐため，偶数と奇数のページを一括して管理する方式もある．ヒット率はページサイズが2倍になるのとほぼ等しいが，TLBミス時のスワップ処理は1ページ単位に行えるので，ページアウト時の時間も1ページごとの管理の場合と変わらない．

昔，プログラムには実行の局所性があるので，TLBミスを発生した前後のページを余分に変換してTLBに入れておくという特許をよく見かけた．しかし，補助記憶装置に対するリード/ライトもその分だけ増加するので，本当に実用的かどうかわからない．

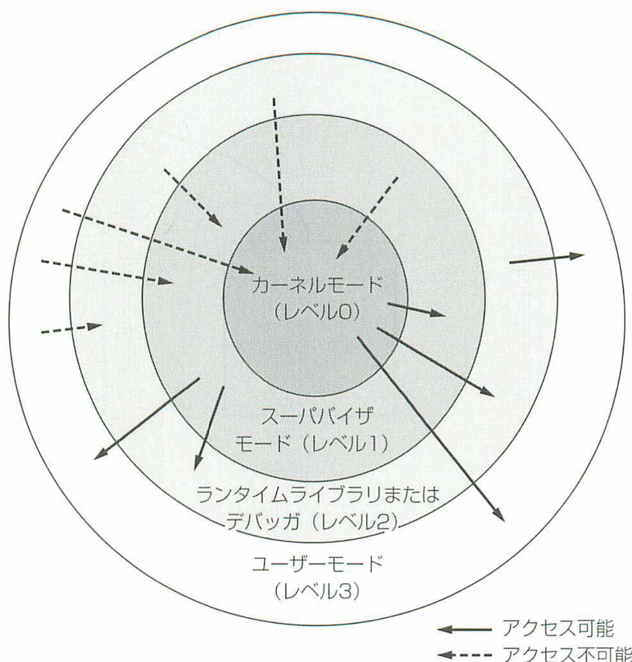


図8 実行レベルとメモリ保護

5 メモリ保護

● 実行レベルについて

悪意のあるアプリケーションプログラム，あるいは，バグのあるアプリケーションプログラムの暴走でOSの領域を壊さないように，MMUはメモリ保護の機能を提供する．上述のように仮想記憶モードではPTEによってリード/ライト属性による保護が実現されている．通常，プログラム領域はリードのみ可，データ領域はリード/ライト可能に設定されている．このほかにもユーザー，カーネルといった特権性による保護が行われる．これについて説明する．

MPUのアーキテクチャでは，プログラムの実行レベルというものが定義されている．これは特権性の強さを表すもので，通常，アプリケーションプログラムは最低の特権性の下で実行される．メモリ保護とは，仮想アドレス空間の各ページに保護レベル(そこのプログラムやデータにアクセスできる最低の実行レベル)をもたせ，特権性の低いプログラムから，より特権性の高いプログラムへのアクセスをできなくする機能である(図8)．

実行レベルの種類は，アーキテクチャによって異なるが，2～4種が定義されている．2～3種の場合，実行レベルに名称が付いていることが多い．4種の場合は，単に，レベル0，レベル1，レベル2，レベル3と呼ぶ．値が小さいほど特権性が高い．たとえば実行レベルの名称は，次のようになっている．

2レベル：カーネル ＞ユーザー
　　　　スーパバイザ ＞ユーザー
3レベル：カーネル ＞スーパバイザ ＞ユーザー
4レベル：レベル0＝
　　　　カーネル ＞レベル1 ＞レベル2 ＞
　　　　レベル3＝ユーザー

ここで，不等号は特権性の高さを表すものとする．カーネルとはOSの実行レベルであり，スーパバイザとはデバイスドライバやランタイムライブラリの実行レベルである．ユーザーとはアプリケーションプログラムの実行レベルである．多くのOSでは，カーネル(あるいはスーパバイザ)とユーザーの2レベルしか使用しない．その中間の実行レベルは，あれば便利だが，OSの構造が複雑になるのであまり使用されない．

通常，実行レベルはMPUのステータスレジスタに格納されている．一方，保護レベルはPTEで指定され，同等の情報がTLBにも格納されている．そして，アドレス変換時に現在の実行レベルとアクセスするページの保護レベルが比較され，自分と特権性が同じか，特権性が低いページであるとアクセスが許可される．アクセスが禁止されている場合はメモリ保護例外やアドレス例外が発生する．

PTEでは，リード，ライトといった，アクセスの種類での保護も可能になっている．つまり，リード可能，

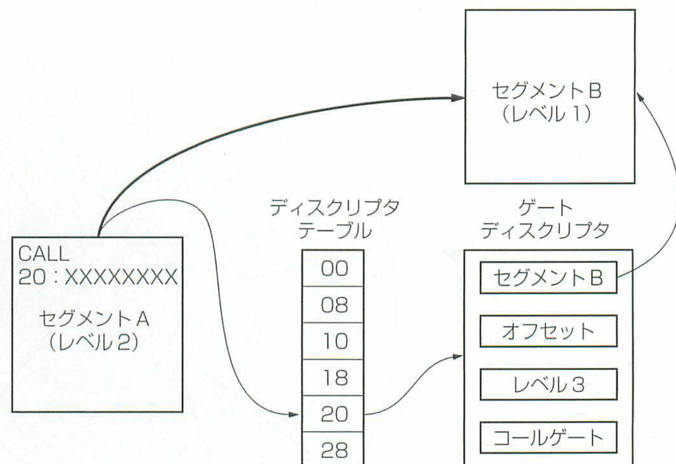


図9　コールゲートの概念

ライト可能，リード/ライト可能といったページ保護を独立に指定できる．アーキテクチャによっては「実行」というアクセスの種類をもっているものもある．

以上は，たいていのMPUの保護機構であるが，もっともシェアの高い(と思われる)x86アーキテクチャでは，少し事情が異なる．ステータスレジスタ(x86でいうところのフラグレジスタ)内に実行レベルを保持しない．現在実行中の仮想アドレス(セグメント)の保護レベルが，そのまま現在の実行レベルとなる．属するセグメントが変わるとき(FAR CALLや例外など)に，移行先の仮想アドレスの保護レベルと現在の実行レベルの比較を行ってメモリ保護を実現する．

● 実行レベルの変更

MPUは，リセット直後は最高の特権性をもっている．そしてアプリケーションプログラムを実行する直前に最低の特権性に移行する．また，アプリケーションプログラムの実行中に割り込みや例外が発生すると，最高の特権性に戻る．

カーネルモードからユーザーモードへの移行は，具体的には割り込みからの復帰命令を利用する．この命令(たとえばERET)はスタックから新しいステータスレジスタの値と新しいPC(プログラムカウンタ)の値をリードして，そのPCの示すアドレスに分岐する．このとき，スタックに積んであったステータスレジスタに設定される値の中に新しい実行レベルが含まれている．アーキテクチャによってはスタックではなく，特殊レジスタからステータスレジスタとPCの値をリードするが，実質は同じである．

ユーザーモードからより特権性の高い実行レベルに移行するには，専用命令が用意されている場合もあるが，通常はソフトウェア割り込み(トラップ命令やシステムコール命令)によって，一律に特権性が最高のカーネルモードに戻ることが多い．専用命令が用意されていない場合，ユーザーモードから特権性が中間のレベルに移行するのは難しく，一度，カーネルモードに移る必要がある．

ただしx86では，また事情が異なる．仮想アドレスでの保護レベルで許可されていれば，コールゲートを使用して任意の実行レベルに移行できる．x86のプロテクトモードにおいて，セグメントレジスタの値はディスクリプタテーブルと呼ばれる，新しいセグメントとオフセットが組になったディスクリプタの集まりへの選択情報となる．コールゲートを呼び出すにはセグメント間コール(FAR CALL)やセグメント間ジャンプ(FAR JUMP)を利用する．新しいセグメントの値で選択されたディスクリプタがゲートディスクリプタである場合がコールゲートとなる．

ゲートディスクリプタには，新しいセグメントとオフセットの値のほかに，そのゲートをコールできる(最低の)実行レベルが格納されている．FAR CALL/FAR JUMPを実行するプログラムが存在しているアドレスの実行レベルがゲートディスクリプタの実行レベルより特権性が高ければ，どの実行レベル(のセグメント)にも移行できる(**図9**)．なお，割り込みや例外が発生した場合は，(一般)ディスクリプタテーブルの代わりに例外ディスクリプタテーブルが参照され，最高の特権レベル(レベル0)に移行する．

● 仮想アドレスによるメモリ保護

ところで，メモリ保護はTLBで行うのが普通だが，仮想アドレスの値そのもので保護を行う場合もある．現



図10　MIPSのアドレス空間（32ビットモード）

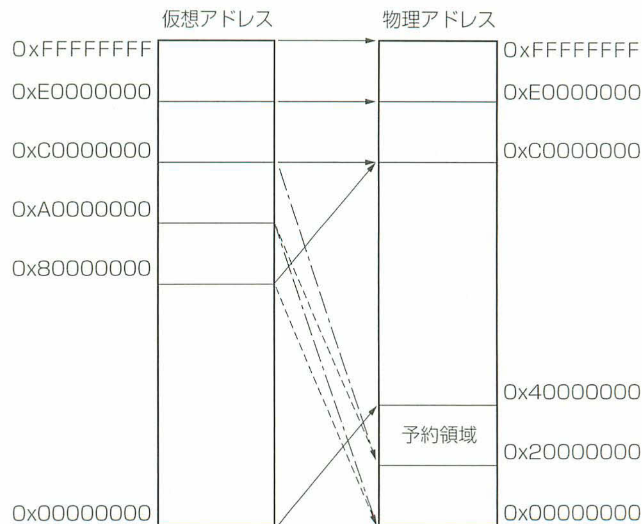


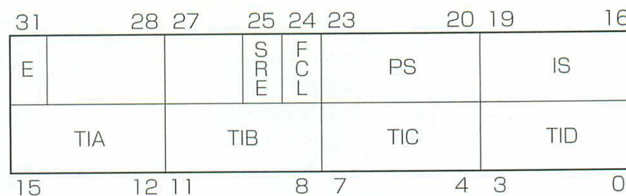
図11　Jade（4Kp）**のBAT**

在の実行レベルに応じてアクセスできる仮想アドレスが最初から規定されているのである．たとえば，MIPSのアーキテクチャがそうだ（**図10**）．ユーザーモードでは仮想アドレスの0x80000000～0xFFFFFFFFの範囲はアクセスできないし，スーパバイザモードでは仮想アドレスの0x80000000～0xBFFFFFFF，0xE0000000～0xFFFFFFFFの範囲がアクセスできない．しかしカーネルモードではすべてのアドレス空間をアクセスできる．

そもそも組み込み制御分野では，アドレス変換が必要ない場合が多い．メモリ保護さえあればよい．このような要求に対応するためにMIPSのJade（4Kp）では，BAT（Block Address Transfer）と呼ばれる機構を提供している（**図11**）．これは，仮想アドレスは基本的に物理アドレスと同じになり，メモリ保護だけは**図10**と同等になる機構である．あるいは，MC680x0では現在の実行レベルやアクセスの種類がファンクションコード（FC2，FC1，FC0）としてMPUの外部端子に出力されている．この信号とアドレスバスの値を外部回路で処理して，保護違反のアクセスを検出すると，バスエラーをMPUに通知できるしくみを提供している．

6 MMUの実例

ここでは，いくつかのMPUでMMUの実例を見て



E：変換許可
SRE：SRP(スーパバイザルートポインタの許可)
　SRE＝0：：変換はすべてCRP(CPUルートポインタ)から始まる
　SRE＝1：：ユーザーアクセスはCRP．スーパバイザアクセスはSRPを使用．
FCL：FC(ファンクションコード)のルックアップ．つまり最初のディスクリプタのアクセスをFCの値でインデックスするか
PS：ページサイズ．1ページのビット数を指定
IS：イニシャルシフト．仮想アドレスのサイズ(32〜17ビット)を指定．つまり，アドレス変換時に仮想アドレスの上位をマスクするビット数．32ビットなら0を指定
TISx：テーブルインデックス．TIAが1レベル目，TIBが2レベル目，TICが3レベル目，TIDが4レベル目．それぞれ仮想アドレスの中で何ビットを占めるかを指定する
注意：IS＋TIA＋TIV＋TIC＋TID＋PS＝32(ビット)の関係を保たなければならない

図12
MC68030の変換制御レジスタ(TC)

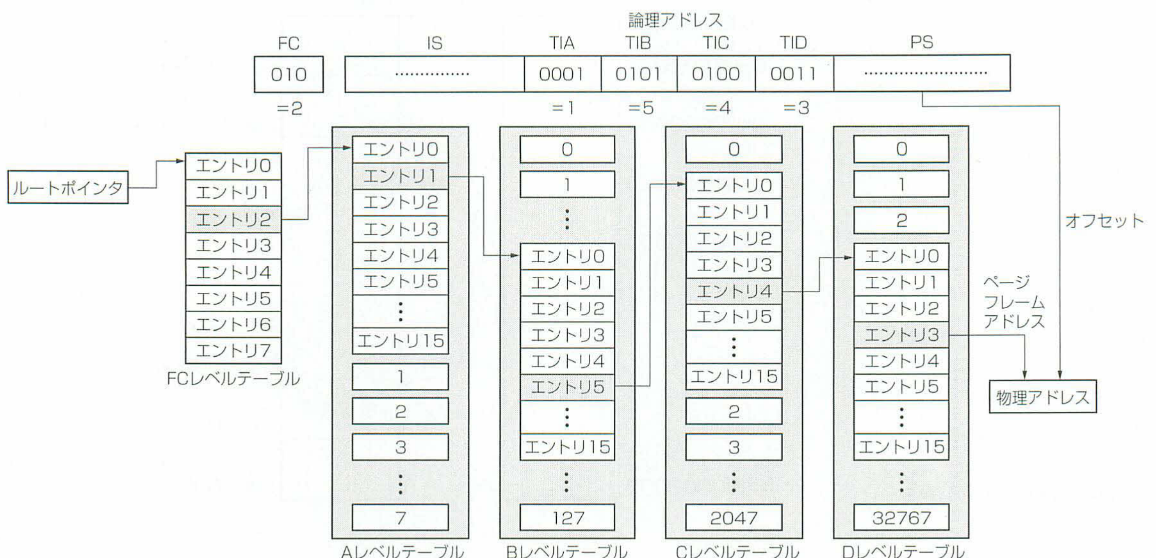


図13　MC68030の5レベルのテーブルサーチ

みよう．

● MC68030/MC68040のMMU

▶ アドレス変換

MC68030ではアドレス変換は0〜5レベルの範囲で自由に設定できる．その設定を行うためのレジスタが変換制御レジスタ(TC)である．**図12**にTCの形式を示す．TIA，TIB，TIC，TIDでそれぞれ1レベル，FC(Function Code＝保護レベル)ルックアップを行えばさらに1レベル増えるので，最大5レベルのページングとなる(インダイレクト指定をすれば6レベルまで可能ということだが，ここでは触れない)．TIA，TIB，TIC，TIDの値を0に設定することで1〜4レベルのページングが可能になる．ゼロレベルというのは，ルートポインタ(CRP，SRP)の中に直接変換後の物理アドレスが指定されている場合(アーリーターミネーションという)である．

MC68030では，TLB(ATC)ミスに際し，

CRP(SRP)→(FC)→TIA→TIB→TIC→TID→

と，変換テーブルをたどっていき(テーブルサーチ)，最終的にページテーブルに到達する．5レベルのアドレス変換例を**図13**に示す．この例ではPSは256バイト(8ビット)で，TIA，TIB，TIC，TIDはそれぞれ4

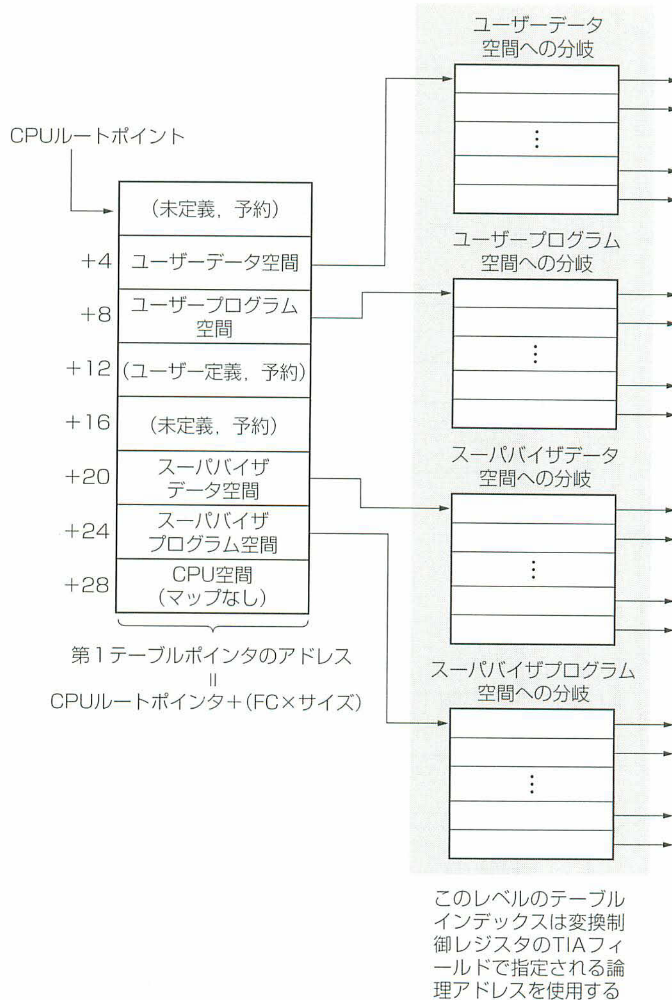


図14　MC68030のFCルックアップを用いたテーブルサーチ

ビット(各テーブルは16エントリ)である．**図14**にはFCルックアップを用いたテーブルサーチ例を示す．FCによって，次にアクセスするレベルAテーブル(TIAでインデックスされるテーブル)のベースアドレスを個別に設定できるので，メモリ保護が実現きる．

MC68030のMMUでは変換テーブルをサーチするために，仮想アドレスを細かく分割しすぎている感もある．実際的には2～3レベルのページングしか行われないので，オーバスペックとも思える．後継のMC68040ではこの点が改良された(退化と呼ぶ人もいるが)．テーブルサーチは3レベルに固定し，ページサイズも8Kバイトまたは4Kバイトのみが許されている．テーブルインデックスはレベルAテーブル，レベルBテーブルが7ビット，レベルCテーブルが5ビット(1ページ8Kバイトの場合)，または，6ビット(1ページ4Kバイトの場合)である．MC68030風にいえば，

IS　= 0
TIA = 7
TIB = 7
TIC = 5(8Kバイト/ページ),
　　　6(4Kバイト/ページ)
PS　= 13(8Kバイト/ページ),
　　　12(4Kバイト/ページ)

ということになる．

▶ATC(TLB)

MC68030のATCは，22エントリのフルアソシアティブ構成である．仮想アドレスとFCを組で検索し物理アドレスを得る(**図15**．ただし，この図は機能から推測した予想図)．一方，MC68040のATCは64エントリの4ウェイセットアソシアティブ構成である．仮

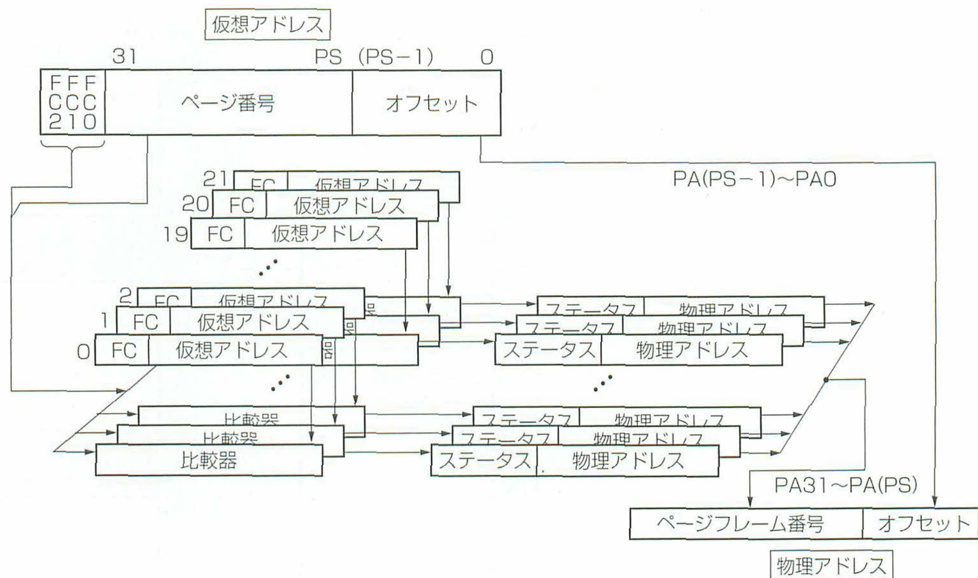


図15
MC68030のATC
(筆者想像図)

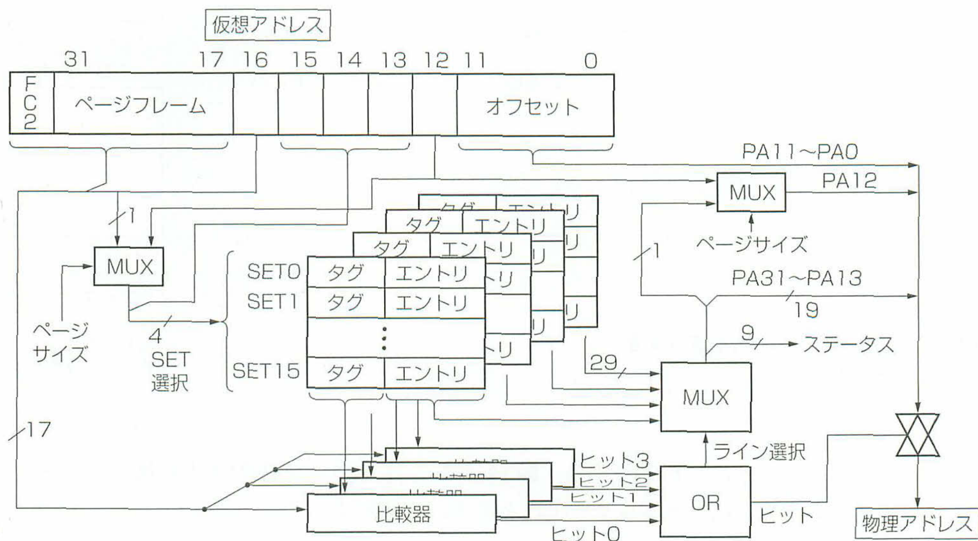


図16
MC68040のATC

想アドレスとFCの最上位ビット(FC[2])を組で検索し，物理アドレスを得る(**図16**)．FCの最上位しか見ないということは，スーパバイザとユーザーを区別するだけで，命令とデータの区別を行わないことを意味する．

● i486のMMU

Pentium以降，IntelのMPUの内部構造に関して詳しく記してある資料は少ない．Intelアーキテクチャはi386で完成しているので，MMUの基本構造もそれ以降大きな変化はないと予想される．ここでは，i486のMMUを**図17**に示す．

TLBは全32エントリの4ウェイセットアソシアティブ構成を採る．セグメントユニットによって生成されたリニアアドレスのビット14〜12でTLBのエントリ(セット)をインデックスし，そこから選択される四つのタグブロックの値と，リニアアドレスのビット31〜15を比較する．もし，どれかと一致すればヒットであり，どれとも一致しなければミスである．ヒットする場合は，そのウェイとセットに対応するデータを物理アドレスのページアドレスとしてアドレス変換を行う．ミスの場合は，MPUはメモリ上のアドレス変換テーブルを検索し，リニアアドレスに対応する変換情報を，TLBの指定されたエントリに格納する．そして，再びアドレス変換を試みる(当然，次は必ず

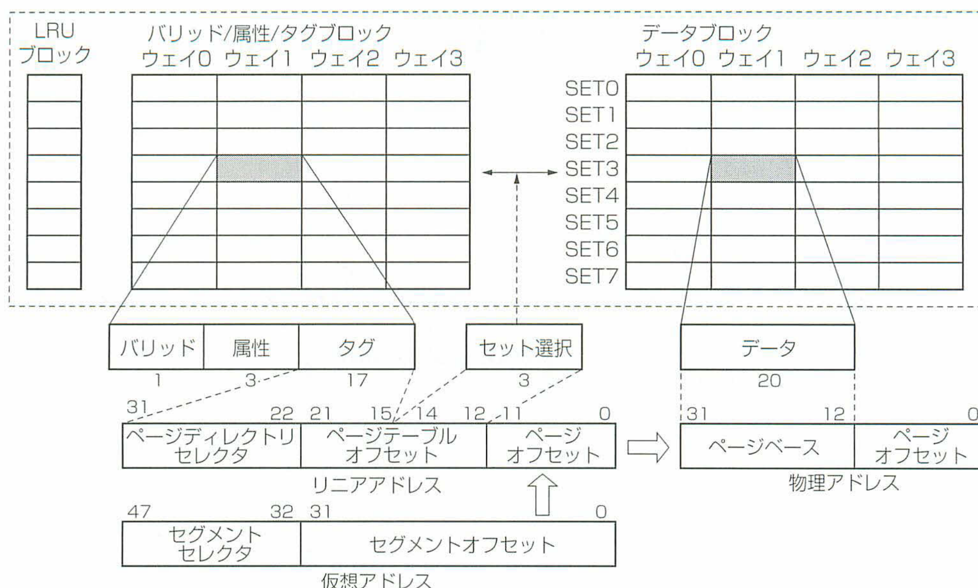


図17
i486のTLB

ヒットする)．このとき書き潰されるウェイは，疑似LRUにより，もっとも参照された頻度が少ないものが選択される．

●Quantispeed(Athlon/Hammer)のMMU

AMDのAthlonやHammerで採用されている，Quantispeedアーキテクチャの一つに排他的，投機的TLBがある．**図18**にHammerのTLBの構成を示す．

TLBはマイクロTLB(L1-TLB)とTLB(L2-TLB)の2段構成を採り，これらの内容は排他的である．TLB自体も巨大で，効率的なアドレス変換ができる．Hammerの命令TLBはL1-TLBが40エントリのフルアソシアティブ，L2-TLBが512エントリの4ウェイセットアソシアティブである．データに関しては，2組のAGU(Address Generation Unit)に対応して2組のL1-TLBがあり，それらは各40エントリのフルアソシアティブである．L2-TLBは512エントリの4ウェイセットアソシアティブである．これらが排他的(Exclusive)であるので，命令は552エントリのTLB，データは592エントリのTLBとして機能する．また，TLBは投機的にリフィルを行い，将来のアドレス変換に備える．

TLBの具体的な挙動については，資料がないので，想像に過ぎないが，次のように機能すると考えられる．

▶L1-TLBにヒットすると，そのエントリでアドレス変換する．

▶L1-TLBにミス，L2-TLBにヒットすると，L2-TLBのエントリでアドレス変換する．このとき，L1-TLBとL2-TLBのエントリを交換する仕様かもしれない(L1-TLBのほうがアクセス時間が短いと思われるため)．

▶L1-TLB，そしてL2-TLBにミスすると，L1-TLBのもっとも古いエントリをL2-TLBに退避し，メモリからL1-TLBにリフィルする．現実的にはL2-TLBにリフィルし，その後L1-TLBのエントリと交換すると思われる．

投機的なリフィルに関しては，1回のTLBミスに対し，その前後のアドレスの変換情報をL2-TLBにリフィルすると思われるが詳細は不明である．L1-TLBから追い出されてくるエントリと投機的にリフィルされるエントリが同一のエントリを占める場合はどちらが優先されるのだろうか．あるいは，L1-TLBからL2-TLBの書き戻しは発生しないのかもしれない．

なお，AthlonではL1-TLBは，命令とデータで各24エントリ，L2-TLBは各256エントリである．つまり，HammerはAthlonの約2倍のエントリをもつ．

●MIPSアーキテクチャのMMU

▶TLBの概要

MIPSアーキテクチャのMMUにはTLBしかない．MPUは仮想アドレスがTLB内にあるか否かを検索し，ヒット(ある)すれば対応する物理アドレスを供給する．ミス(ない)あるいは保護違反を検出する場合はTLB例外を発生するのみである．

TLBミスが発生してもアドレス変換テーブルを自動的に検索し，TLBのエントリを入れ替えるという

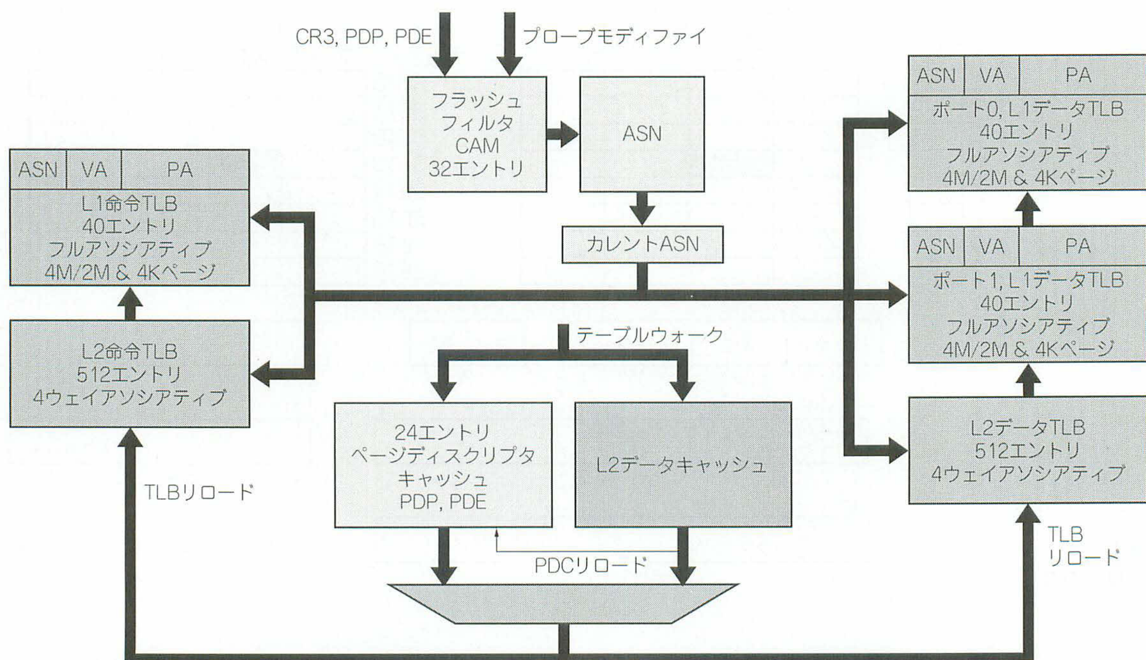
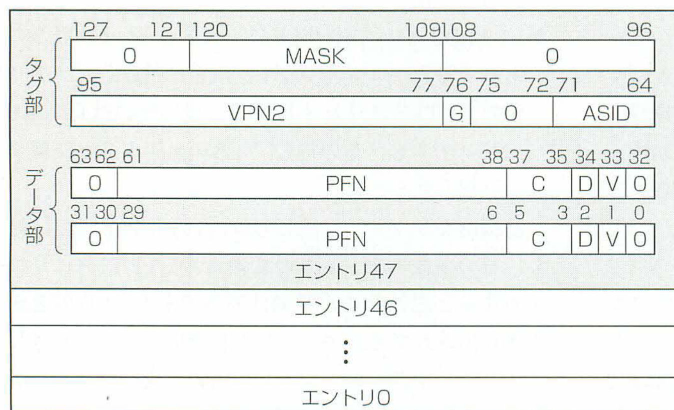


図18　HammerのTLBの構成



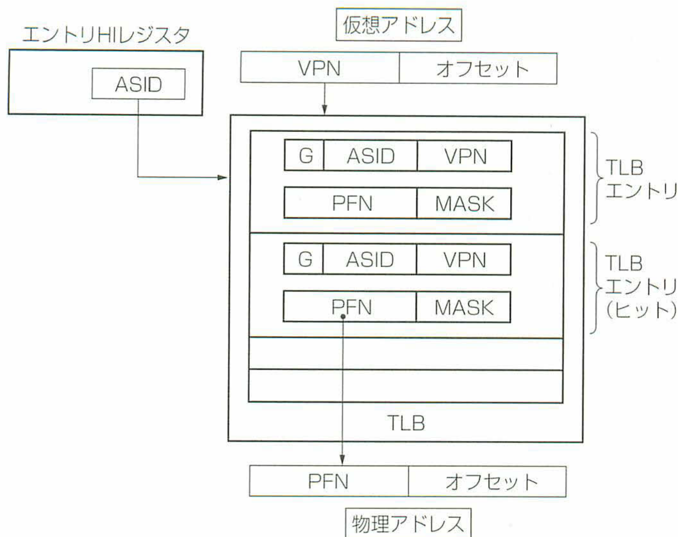
MASK：ページ比較マスク，VPN2：仮想ページ番号，G：グローバル(0のとき，TLBルックアップ時にASIDを比較)，ASID：アドレス空間ID：PFN：ページフレーム番号(物理ページの上位ビット)，C：キャッシュアルゴリズム，D：ダーティ，V：バリッド

図19　MIPSのTLB(R4000/32ビットモード)

操作は行わない．代わりにTLBの内容をソフトウェアで操作できるようになっており，TLBミス発生時のエントリ入れ替え処理はソフトウェアで行うことになっている．エントリ入れ替えのための複雑なハードウェアは省略するというRISCならではの考え方である．MIPSアーキテクチャの発表当時，TLBの更新は数命令で実現可能であり，システム性能の低下はないと明言されていた．

R4000以降，MIPS系のMPUは64ビットプロセッサであり，アドレス空間に関して32ビットモードと64ビットモードをもっている．TLBの各エントリも32ビットモードと64ビットモードで若干異なる．**図19**にTLBエントリの形式を示す．各エントリは特権レジスタである，エントリHi，エントリLo0，エントリLo1，ページマスクレジスタと直接対応する領域をもっている．

TLBは，32ビットモードにおいては32ビットの仮想アドレスを，64ビットモードにおいては64ビットの仮想アドレス(TLBには40ビット分の領域しかないが)を，通常は36ビットの物理アドレスに変換する．このしくみを**図20**に示す．物理アドレスのビット数はMPUによって異なり，それによってエントリLo0,



- エントリHIレジスタには現在のアドレス空間
- ID(タスクID)を格納しておく仮想アドレスのVPN(ページ番号)とASIDがTLBの各エントリのVPN，ASIDと比較される．ASIDを比較するのはGビットが0の場合のみ．比較時にMASKビットでVPNの下位ビットをマスクすることで種々のページサイズに対応する
- 比較内容が一致すると，PFN(物理アドレスの上位ビット)を取り出す
- オフセット部分がTLBを通さずに使用され，PFNと結合して物理アドレスを生成する

図20　MIPSのアドレス変換

エントリLo1レジスタ内のPFN領域のビット数が決定されている．

MMUのサポートするページサイズは，エントリごとに4Kバイトから16Mバイトの範囲で4の倍数ごとに指定できる．これは，エントリへの書き込み時にページマスクレジスタで指定する．アドレス変換時に，仮想アドレス番号(VPN)の下位ビットをページマスクレジスタの値で無視して仮想アドレスの検索(一致比較)が行われる．

▶ タブルエントリ構成

MIPS系のMMUの大きな特徴は，二つのページを組にして扱う点である．TLBは48エントリ(R4200，R4300では32エントリ)のフルアソシアティブ構成で，1エントリは連続する2ページ分(偶数ページと奇数ページ)を示す一つの仮想アドレスと，それに対応する二つの物理アドレスを保持している．

仮想アドレスはエントリHiレジスタ，偶数ページ/奇数ページに対応する物理アドレスは，それぞれ，エントリLo0レジスタ/エントリLo1レジスタで指定する．このTLB形態は一般にダブルエントリ形式と呼ばれている．48エントリではあるが，実質的には，98エントリ相当，あるいは指定したページサイズの2倍のページサイズをもっているとみなせるためTLBのヒット率が高くなるといわれている．

▶ ASID

なお，エントリHiレジスタは仮想アドレスのほかにタスク番号に対応するASID(Address Space ID)を指定できる．TLBの各エントリにもASIDに対応する領域があり，仮想アドレスの検索時にASIDの一致も調べられる．仮想アドレスとASIDが一致して初めてヒットとなる．このため，マルチタスク環境下においてタスク切り替えが生じても，エントリHiレジスタのASIDを変更してさえおけば，TLBエントリの無効化をする必要がない．

たとえば，あるタスクの仮想アドレス0番地と，他のタスクの仮想アドレス0番地に対応する物理アドレスは一般には異なるので，ASIDがなければ，0番地を他のタスクの物理アドレスに変換してしまう必要がある．このため，ASIDをもたないTLB構成においては，タスク切り替え時にTLBの全エントリを無効化しておく必要がある．OSやライブラリ空間など各タスク間で共有する領域に関しては，TLBエントリのグローバルビットをセットしておけばよい．この場合，仮想アドレスの検索はASIDを無視して行う．

また，エントリLo0，エントリLo1レジスタは，対応するページのキャッシュ情報，保護情報も指定できるようになっている．

▶ 入れ替え方式

MIPSアーキテクチャでは，TLBミスが生じた場合，エントリの入れ替えはソフトウェアで行う．どのエントリを追い出すかはランダム(任意)に決定する．といっても実際には，ランダムレジスタという特権レジスタが指し示すエントリを更新することになる．ランダムレジスタは，ワイヤードレジスタ(特権レジスタ)で示される値と(TLBエントリ数－1)の間の任意の値を保持している．つまり，ランダムレジスタは0から

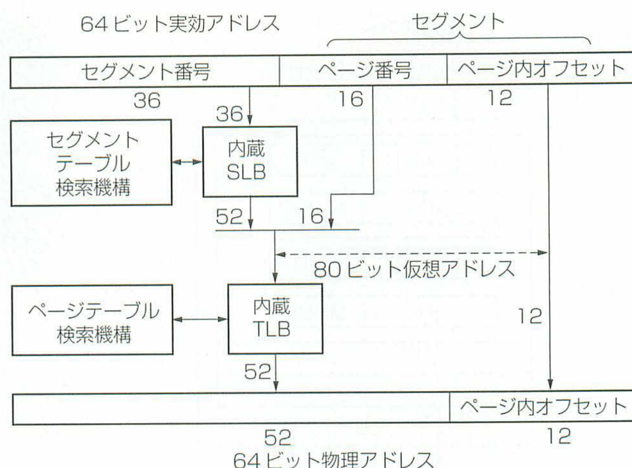


図21
PowerPC(64ビットモード)のMMU

(ワイヤードレジスター1)の値は指し示すことがないので，TLBのエントリ0から(ワイヤードレジスター1)までは決して追い出されることのない安全な(必ずヒットする)エントリとして確保できる．

● PowerPC(64ビットモード)のMMU

2002年秋のMicroprocessor Forumで，IBMからPowerPCの64ビット実装であるPowerPC 970が発表された．現在ではPowerMac G5に搭載されているMPUである．

64ビットPowerPCといえば，かつてPowerPC 620が計画されたが実現には至らなかった．しかし，マイクロアーキテクチャを少し変更してPower3として登場した．PowerPCの64ビット実装の最初は，1995年に登場したA30と呼ばれるAS/400用のMPUである．A30は1個のCMOSチップと6個のBiCMOSチップの合計7チップからなるPowerPC唯一のマルチチップ実装であえる．A30は1997年にシングルチップ実装のPowerPC RS64に置き換えられた．その後RS64-Ⅱ，RS64-Ⅲ，RS64-IVと改良が続けられ，AS/400，RS/6000 S80シリーズなどのビジネス用サーバとして利用されている．A30はサーバ用なので，本来のPowerPCとはいえない．

さて，PowerPCの64ビット実装では，マシン状態レジスタのビット指定により，64/32ビットモードを切り替えることができる．セグメントサイズとページサイズは32ビットモードと同様で，それぞれ，256Mバイトと4Kバイトである．このため，64ビットモードではセグメント数(16個から64G個)とセグメントのビット幅の拡張(24ビットから52ビット)により仮想アドレス空間を実現する．

なおPowerPCでは，ユーザーが使用する64ビットの仮想アドレスを実効アドレスと呼び，システムが管理する80ビットの仮想アドレスと区別している．アドレス変換機構により，80ビットの仮想アドレスが64ビットの物理アドレスに変換される．

PowerPCでは実効アドレスの上位4ビットでセグメントレジスタを選択していたが，64ビットモードでは実効アドレスの上位36ビットで選択する．36ビットといえば64G個と膨大な数からの選択となるため，すべてを主記憶にもっていたのでは主記憶があふれてしまう．そこで，主記憶上にページテーブルと同様な形式のセグメントテーブルを置き，MPU内部にTLBと同様のSLB(Segment Look-aside Buffer)を内蔵してアドレス変換を行う．アドレス変換時に実効アドレスがSLBに存在しない場合，TLBミスの発生時と同様にセグメントテーブルが自動的に検索されてSLBのエントリを置き換える．

図21に64ビット実装時のPowerPCのアドレス変換を示す．

まとめ

MMUというものが，だいたいどのような働きをするものか理解していただいただろうか．以上をおさえておけば，基礎知識としては十分である．個人的にはx86アーキテクチャに思い入れはないが，図らずもx86のアーキテクチャの説明がかなりの部分を占めてしまった．アドレス変換やメモリ保護についてはx86のやり方は特異にみえるが，現在のコンピュータアーキテクチャを語るうえではこれも必須な教養であろう．

Column 2 セグメント方式

● セグメンテーションの概念

本書では仮想記憶の方式として，ページを単位とするページング方式を中心に説明してきた．ここで，もう一つの主要な仮想記憶方式であるセグメント方式について説明しておこう．これは，セグメントを単位とするのでセグメンテーションともいう．

セグメント方式はプログラムのモジュール分割と関連付けて説明されることが多い．プログラムというものは一つのメインルーチンといくつかのサブルーチンの集まりである．マルチタスクを考えるとき，あるサブルーチンや(ときにはメインルーチンを)共通に使用すれば，メモリの使用効率が向上する．セグメント方式ではメモリ割り当てをメインルーチンやサブルーチン単位に行うことを考える(**図A**)．

セグメント方式ではアドレスの指定を，ベースアドレス(開始アドレス)とベースアドレスからのオフセット値で行う．そして，すべてのメインルーチンやサブルーチンといったモジュールは，オフセット0から開始され，データのアクセスもオフセットで指定するものと仮定する．こうすることで，そのモジュールはメモリ内のどこに配置しても実行可能になる．つまり，リロケータブル(再配置可能)となる．このため，モジュールごとに物理アドレスでベースアドレスを決定してやれば，同じオフセットを有する別のモジュールをメモリ内の自由な位置に置くことができる．モジュール自体は自身がメモリのどこに割り当てられるかを意識する必要はない(**図B**)．

セグメント方式では，仮想アドレスは，(物理アドレスで示される)ベースアドレスを直接/間接的に指定するセグメント値と，セグメント内のオフセット値という二つの情報で規定される．このため，セグメント方式のアドレスは2次元アドレスとも呼ばれる．一方，これまで述べてきた一つの情報で仮想アドレスを指定する方式のアドレスは，1次元アドレス，または線形アドレス(リニアアドレス)と呼ばれる．

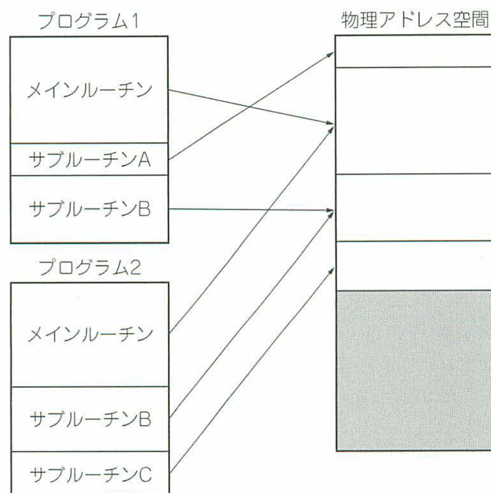
● セグメント単位でのスワップ

さて，大きなプログラムでは，コード部，データ部，スタック部が，それぞれいくつものセグメントに分かれている．このうち，ある時点のプログラムの実行に必要なセグメントのみをメモリに置いて実行すれば，物理メモリの容量を越えるプログラムを実行することもできる．これはメモリスワップの単位がセグメントになっただけで，ページングによる仮想アドレス方式と同じ効果を生む．

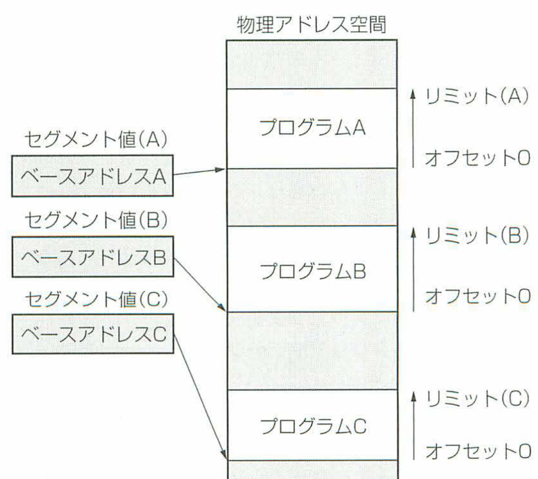
セグメント方式では，ページングでの変換テーブルに相当するものが，セグメントテーブルである．セグメントテーブルの各エントリは，メモリ保護情報，セグメント長(アドレスの上限)，ベースアドレスといった情報を含む．**図C**にセグメント方式でのアドレス変換を示す．MMUの動作としては次のようになる．

1) 仮想アドレスに含まれるセグメント値でセグメントテーブルをアクセスする
2) アクセスされたセグメントテーブルのエントリからベースアドレスを得る
3) ベースアドレスとセグメント内オフセットを結合して物理アドレスを生成する

x86アーキテクチャにおいて，セグメント値はセグメントレジスタに格納されているので，直接仮想アドレスの一部としては見えない．さらに，リアルモードにおいては，セグメント値を4ビット左シフトしてベースアドレスとしている．

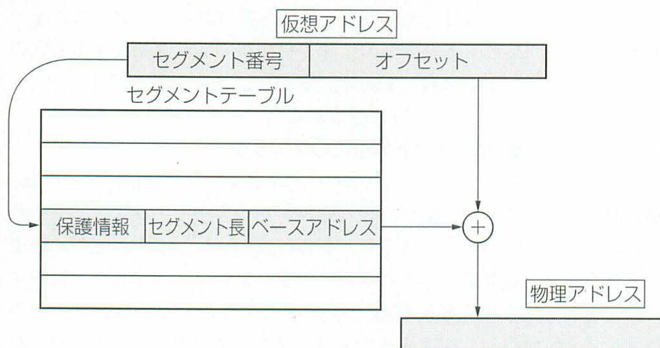


図A　セグメンテーションの概念



図B　セグメンテーションの概念

Column 2 セグメント方式(つづき)

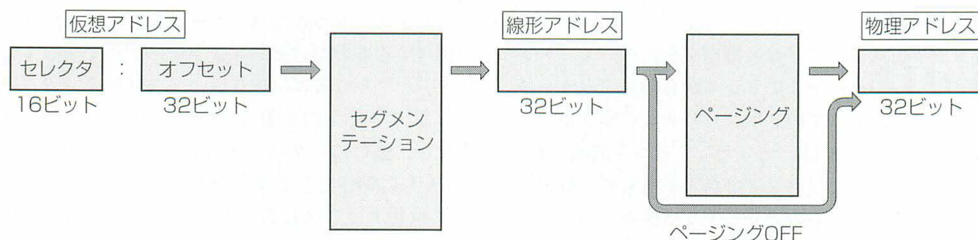


図C　セグメンテーションのアドレス変換



INDEX：GDT/LDTへのインデックス
TI：テーブルインデックス
RPL：要求レベル

図E　セグメントレジスタ



図D　セグメンテーション+ページング(x86)

● x86でのセグメンテーション例

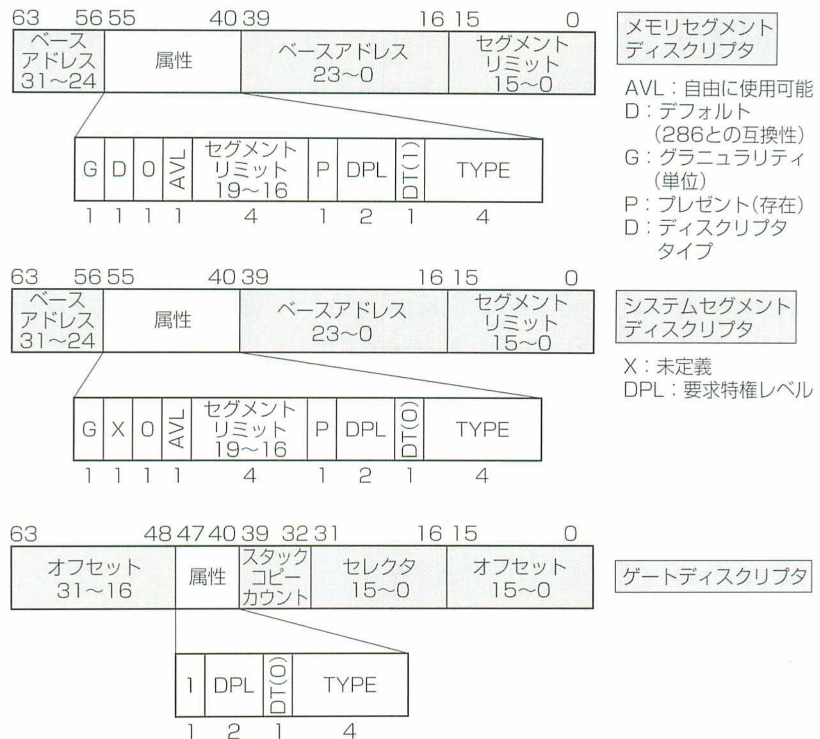
x86アーキテクチャ(プロテクトモード)では，仮想記憶のアドレッシングにセグメンテーションとページングを併用している．図Dに示すように，セレクタ値(セグメント値)とオフセットからなる2次元アドレスがセグメンテーションによって線形アドレスに変換され，それを仮想アドレスとしてページングを行って物理アドレスを得る．

セレクタとはセグメントテーブルへのインデックス，セグメントテーブルの種類，要求特権レベルという三つの領域からなる16ビットの情報である．図Eにセレクタを示す．x86では実行中のセグメントの保護レベルが現在の実行レベルになることはすでに述べたが，セレクタ中の要求特権レベルは実行レベルの特権性を下げる効果がある．つまり，現在の実行レベルと要求特権レベルを比較して特権性が低いレベルのほうが現在の実行レベルとみなされる．通常は，レベル0(最高の特権性)となっている．

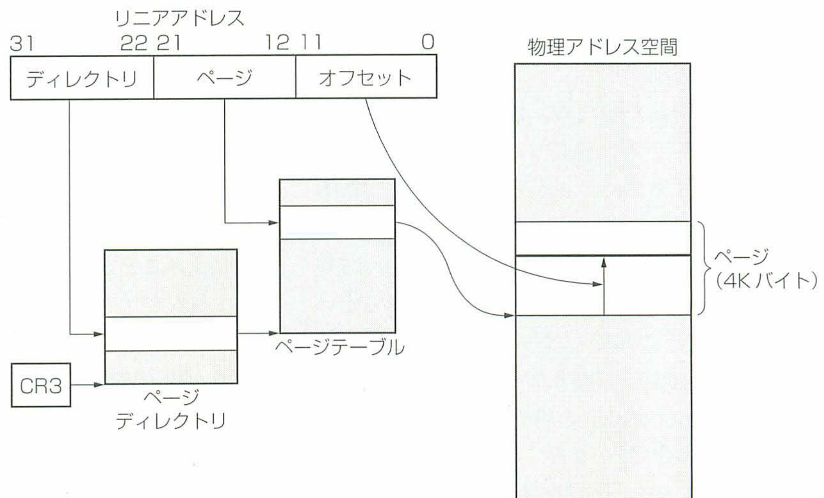
● グローバルディスクリプタテーブルとローカルディスクリプタテーブル

セグメントテーブルにはグローバルディスクリプタテーブル(GDT)とローカルディスクリプタテーブル(LDT)の2種類がある．GDTとは，システム内に一つだけ存在するセグメントテーブルのことで，OSや複数のタスクから共通にアクセスされるメモリ領域を定義する．それに対しLDTは，タスクごとのメモリ領域を定義する．そして，セレクタのインデックスは，GDTまたはLDT内のエントリ(それぞれをセグメントディスクリプタと呼ぶ)を選択する．テーブルインデックスは，このセグメントのディスクリプタがGDTであるかLDTであるかを示す．

GDT/LDTの各エントリは，セグメントディスクリプタと呼ばれる．これは，32ビットのベースアドレス，20ビットのリミット値，その他の情報から構成される64ビットのデータである．図Fにセグメントディスクリプタを示す．セグメントディスクリプタのうち，G(Granularity)ビットはリミット値の単位を指定する．G=0なら単位は1バイトであり，セグメントの大きさは0～1Mバイトとなる．G=1なら単位は4Kバイトであり，セグメントの大きさは0～4Gバイトとなる．DPL(Descriptor Privilege Level)はそのセグメントの保護レベルである．DT(Descriptor Type)はセグメントディスクリプタの示すセグメントの種類(メモリセグメント，システムセグメント，ゲート)を指定する．メモリセグメントかシステムセグメント(またはゲート)かによって，セグメントディスクリプタのTYPE領域の意味が変わってくる．メモリセグメントではリード/ライト/実行の保護情報を指定する．システムセグメントではLDTまたはTSS(Task State Segment)という情報を指定する．ゲートではゲートの種類を指定する．



図F
セグメントディスクリプタ



図G
リニアアドレスから物理アドレスへの変換

なお，メモリセグメントディスクリプタとシステムセグメントディスクリプタにはセグメントのベースアドレスが格納されている（オフセットは線形アドレスのオフセットと同じ）のに対し，ゲートディスクリプタにはポインタ（セレクタ値とオフセット値）が格納されている．ところで，セグメントディスクリプタのベースアドレスやゲートディスクリプタのオフセットが下位と上位に分離して格納されているのは80286との互換性のためである．

セグメンテーションの後はページングが行われるが，これは他のMPUと同様な機構なので詳細な説明は省略する．32ビットの仮想アドレスのビット22～31をディレクトリという単位として1レベル目，ビット12～21をページ単位として2レベル目のテーブルを引き，計2レベルのアドレス変換を行う（**図G**）．

Appendix 2

携帯機器ではとくに重要な 低消費電力技術の原理

「消費電力＝性能」と考えられていたのは昔の話である．現在ではEWSでさえも低消費電力を考慮している．つまり，低消費電力で高性能という要求が強い．

低消費電力の利点は，システム構成の簡略化にある．電力が大きいと熱を発生する．すると熱により装置が誤動作するのを防ぐため，冷却機構の考慮が必要になる．あるいは電源の問題もある．消費電力が大きいと，それに見合う電源供給が必要になるので，強力な電源装置が必要になる．当然，システムの規模も大きく価格も高くなる．

これはパソコン(PC)の世界でも同様である．マザーボードから冷却ファンを取り除きたいというのが，第1の目的である．システム簡略化の次の目的は，携帯性の向上である．ノートPCでは電池寿命の長さが要である．そのためには，できるだけ消費電力が少ないことが必須となる．このためには，MPUだけでなく，周辺機器も低消費電力化する必要がある．MPUはその第一歩である．

これらの要求を満たすため，1999年頃から低消費電力モードの採用がPC用MPUの売りの一つになってきている．たとえば，CrusoeのLongRun，AMDのPowerNow!やIntelのモバイルPentium ⅢのSpeed Stepなどである．AMDの発表ではSpeedStepは7W程度，PowerNow!は3W程度の電力が節約できるという(2000年当時)．これでもけっこう画期的な値で，電力は従来の1/10以下になり，それにより電池寿命が10～20％(SpeedStepの場合)，あるいは30％(PowerNow!の場合)長くなる．しかし，性能低下も著しく，低消費電力モードは無効化して使用する可能性が高いというのが，もっぱらの予測だった．

現在，低消費電力技術をとくに必要としているのは携帯電話やPDAの分野である．この分野では電池駆動が常識であり，1～15日程度の電池寿命を実現するために，10mW～200mWの超低消費電力が要求される．この要求を満たすのは，もはやPC用のMPUでは不可能であり，ARM，MIPS，SHといった新しいアーキテクチャをもつMPUの存在する意義がここにある．

ここでは，低消費電力を実現するための基本原理について説明する．

● 低消費電力の基本原理

従来，いろいろな低消費電力技術が考案されてきているが，その基本原理はただ二つである．駆動電圧を下げることと動作周波数を下げることである．

電力は駆動電圧と消費電流の積で計算される．つまり感覚的には，

(電力)＝(電圧)×(電流)

＝(電圧)×(電圧)/(抵抗値)

……オームの法則

であり，電圧の2乗に比例する．電力を下げるために駆動電圧を下げることは有効である．電圧が1/2になれば電力は1/4になる．

半導体の製造技術の向上につれて微細化が進み，低電力によるトランジスタ駆動が可能になってきている．たとえば，製造プロセスが0.25 μmで2.5V，0.18 μmで1.8V，0.13 μmで1.5V，0.10 μmで1.0Vである．つまり，製造プロセスを進化させることで，自然と電力も下がっていく．

また，電力は動作クロックの周波数(動作周波数)に比例することがわかっている．最近のMPUを構成するトランジスタはCMOS回路である．CMOS回路は，スイッチング時の状態遷移時にのみ電流が流れる．つまり，出力が変化するときのみ電流が流れる(**図A**)．これは状態をできるだけ変更しなければ電力が低減できることを意味する．

ところで，電気信号は動作クロックに同期して切り替わるので，動作周波数が高いほど電流値が大きくなる．つまり，動作周波数を下げることで，動作電流を下げることができ，結果として電力を下げることができる．

以上をまとめていうと，CMOS回路では一般に，

(電力)＝Σ(寄生容量)×(電圧)×(電圧)×(周波数)

という関係が成立することが知られている．これを示すCV^2fという表現はよく使われるので覚えておこう．

LongRun，SpeedStep，PowerNow!はどれも駆動電圧と動作周波数を動的に制御する技術であり，どのようなタイミングで制御するのかが，実現方法のキーポイントとなっている．

たとえば，CrusoeのLongRunは，電圧と周波数の組み合わせを複数用意し，MPUの負荷によって，それらを動的に変更するようになっている．負荷が軽い場合，まず，周波数を低下させ，その後，電圧を低下させる．ここでいちばん工夫を要するのが負荷の状態を判断する方法である．これは，アプリケーションプログラムが発行するコードモーフィング要求の頻度を監視することで実現していると思われる．しかし，その詳細な手法は公開されていない．

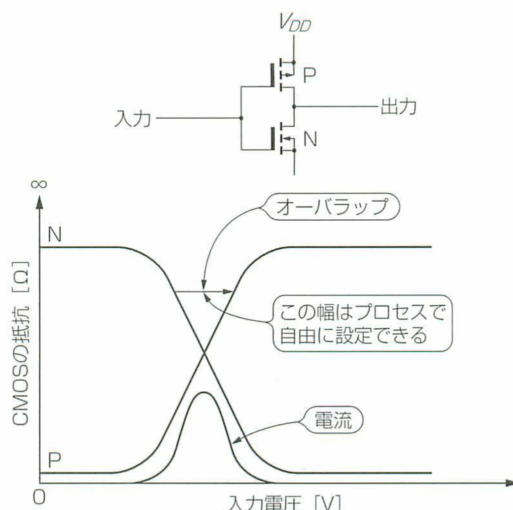
ここで，消費電力に関して興味深い報告を一つ．一般にMPUの動作周波数を上げると消費電力も増大する．しかし，実際のシステムにおいては必ずしも正しくない．動作周波数が速いと周辺回路が動作する時間が短くなるため，結果として消費電力が少なくなる場合もある．ただし，これは周辺機器が同時動作している通常動作時の消費電力の話である．長い期間の電池寿命に効いてくるのは，主として，スタンバイやサスペンドなど待機時の消費電力である．

● **動作電圧の動的制御**

この方法は単純である．供給する電源電圧を上げ下げするだけである．当然，レギュレータなどの外部回路の補助が必要である．また，MPUが負荷状況を外部に通知する手段が必要である．これが，電圧を上げ下げするタイミングとなる．

IntelのモバイルPentium Ⅲで採用されたSpeed Stepは複雑な制御をしない．AC電源を外したときのみ，自動的にクロックと電圧を2段階に制御する．650MHzまたは600MHz時は1.6Vで動作し，500MHz時は1.35Vで動作することで，消費電力を削減する．2001年7月に公開されたEnhanced SpeedStepでは，ソフトウェアで明示的にモード切り替えを行うことも可能である．これはモバイルPentium4-Mで採用された．

TransmetaのCrusoe(TM5400)では，電力管理用に5本の制御信号があり，32段階の電力制御が可能である．これは，主として，電圧制御用である．電圧は1.1V～1.6Vの間で0.05V刻みに変化できる．一方，動作周波数は自動的に変動する．200MHz～700MHzの間を33MHz刻みで変化する．MPUの負荷の変動は0.5 μsごとに検出可能であるという．電圧を変更するのに要する時間は20 μs以内で，アプリケーションの実行に影響を与えることはないといわれている．



図A　CMOSのNOT回路の動作

AMDのK-6+，K6-Ⅲ +，モバイルAthlon4，モバイルDuronなどに採用されているPowerNow!も同様の技術である．そのオートマチックモードでは，MPUの負荷を自動的に判断して電力制御を行う．その具体的な手法に関しては公表されていない．動作電圧は，2.0Vから1.4Vの間を数段階に分けて変更する．動作周波数の変更は，バスクロック(FSB)に対する倍率を変更することで実現する．PowerNow!は，Long Runと比較すると，動作周波数，動作電圧ともに変化の刻み幅が大きい．この意味で，PowerNow!は，SpeedStepとLongRunの中間的な技術といえる．

従来は，電圧の切り替えは2段階で十分としていたIntelも，Pentium M(Banias)ではGeyserville-Ⅲ(ガイザービルⅢ)という技術で多段階の電圧切り替えをサポートするようになった．0.85V時では600MHz動作，1.35V時は1.6GHz動作が可能である．この間で電圧と動作周波数の組み合わせを指定できる．現段階では1GHzという中間値が示唆されている．また，Speed Step(Geyserville-Ⅱ)とは異なり，Geyserville-ⅢではCPU自体が負荷状況をモニタして，自動的に周波数と電圧の組を切り替えるようだ．つまり，ソフトウェアの変更をしなくても，自動的に省電力制御を行うことができる．

結局，LongRun，PowerNow!，Geyservilleは同じようなシステム仕様に近づきつつある．

表A　電力モード

モード	PLL	内蔵周辺ユニット				CPUコア
		リアルタイムクロック	割り込み制御ユニット	バス制御ユニット	その他	
フルスピード	ON	ON	ON	ON	選択可能*	ON
スタンバイ	ON	ON	ON	ON	選択可能*	OFF
サスペンド	ON	ON	ON	OFF	OFF	OFF
EXサスペンド	OFF	ON	ON	OFF	OFF	OFF
ハイバネート	OFF	ON	OFF	OFF	OFF	OFF

ON：クロックを供給する　　OFF：クロックを停止する　　*：CMU（クロックマスクユニット）で選択

(a) V_R4131の電力モード

モード	コア	メモリ	電力	Runモードへの復帰
Run	クロックON 電力ON	クロックON 電力ON	アプリケーション依存	
Standby	クロックOFF 電力ON	クロックOFF 電力ON	リーク電流	割り込みデバッグ要求
Dormant	クロックOFF 電力OFF	クロックOFF 電力ON	メモリからのリーク電流	ソフトリセット
Shutdown	クロックOFF 電力OFF	クロックOFF 電力OFF	ほぼゼロ	リセット

(b) ARM11の電力モード

モード	コア	URAMレジスタ	I/O	電力	Runモードへの復帰
Active	電力ON	電力ON	電力ON	～150mA	
R(Resume)-Standby	電力ON	電力OFF	電力ON	～100μA	割り込み
U(Ultra)-Standby	電力OFF	電力OFF	電力ON	ほぼゼロ	パワーオンリセット

(c) SH-Xの電力モード

● 動作クロックの制御

最近は，ほとんどすべてのMPUがPLL(Phase Locked Loop)を内蔵し，PLLからクロックが供給される．PLLは，原発振(FSBクロックなど)を逓倍することで，安定した高い動作クロックを生成する．動作クロックの制御は，PLLの逓倍率を変更すること，または，PLLの出力を分周することで実現できる．しかし，この手法はMotorolaが特許を取っており，特許侵害を避けるため他社製MPUでの実現方法は意図的に不明確になっていることが多い．

(1) 静的制御

これは，携帯電話の待ち受け時など，MPUが高速で動作する必要がないことが分かっている場合，ソフトウェアによって明示的に動作クロックを分周して低下させる方法である．専用命令の実行，あるいは，レジスタ設定によってクロックの分周比を変更する．

(2) 動的制御

これは，ハードウェアによってMPUの負荷状況を監視し，動作クロックを変動させる方法である．TransmetaのLongRunやAMDのPowerNow!で実現されている．

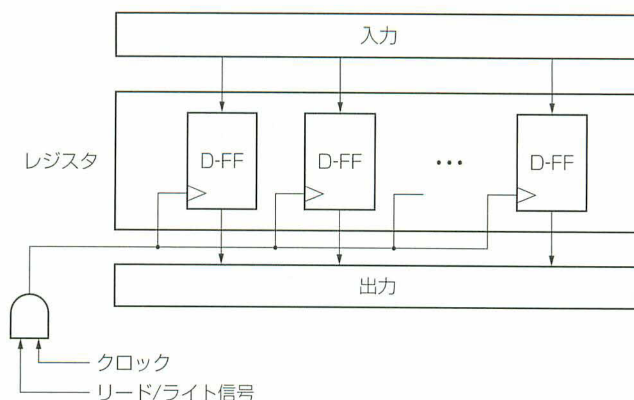
MPU内のハードウェアによる監視を行わなくても，専用の入力端子を用意して実現する方法も考えられる．もっともこれは，監視回路をMPU外部に追い出したのと同じことであるが．

● クロック供給の制御

通常，MPUはいろいろなユニットの集合体で実現されている．そこで，MPUの動作に必要なユニットにのみクロックを供給する方法がある．

(1) 静的制御

これは，低電力モードとして専用命令で提供されることが多い．低電力モードに移るとMPUは待機状態に入る．そして，割り込みが入ると待機状態から抜け出す．低電力モードには，クロックを供給する程度に応じて，スタンバイ，ウェイト，サスペンド，ハイバーネートなどと固有の名称が付けられている．たとえば，NECのV_R4131は**表A(a)**に示す五つの電力モードをもっている．これらの動作モードには，STANDBY，SUSPEND，HIBERNATEといった命令を実行することで移行する．また，**表A(b)**に示すように，ARM社のARM11でも同様の電力管理を実現している．



図B　ゲーテッドクロック

また，MPUによってはクロック制御ユニットをもつものもある．これは，各ユニットへのクロック供給を，ソフトウェアによって明示的に制御しようとするものである．MPUに備わっている機能ユニットでも，アプリケーションによっては，まったく使用しないユニットがある．これらのユニットにクロックを供給するのはむだなので，明示的に動作を止めようという考え方である．ルネサステクノロジ(旧：日立製作所)の携帯電話向けSH-Mobile(SH3-DSPをコアとする省電力プロセッサ群)では，自動的に(？)，各周辺ユニットに供給するクロックを停止できるようだ〔表A(c)〕.

(2) 局所的な動的制御(ゲーテッドクロック)

低消費電力に関しては，電圧や周波数だけでなく，MPUのシステム設計の段階で，フリップフロップ単位に低電力のしくみをもたせる方法もある．これは，マスククロックまたはゲーテッドクロック(あるいはクロックゲーティング)と呼ばれる手法で，必要な場合にしかフリップフロップにクロックを供給しない技術である．フリップフロップ内部では，クロックの変化に応じて状態を更新することで電力を消費する．このクロックを，CMOS回路の状態遷移に必要な最小限の期間しか動作させないという技術である．

具体的には，多ビットのレジスタやバッファに対し，選択信号(リード信号やライト信号)が出力されているときのみ，クロックを供給する(図B)．もっとも，リードの場合はフリップフロップの値を読むだけ(状態を変えない)なので，クロック供給の必要はない．また，1ビット程度のレジスタ(フリップフロップ)では，クロックマスク回路の規模のほうが大きくなるので，意味がない．現状，とくに低消費電力を意識する場合は，3～4ビット程度以上のレジスタの場合は，ゲーティッドを行うようである．

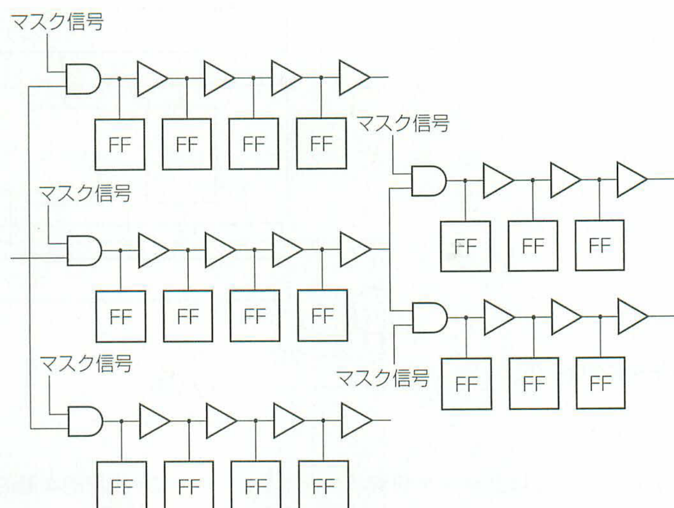
(3) 大局的な動的制御(ゲーテッドCTS)

MPUの内部回路にクロックを供給する場合，場所によってクロック間の遅延がないようにすることは非常に重要である．そのために，類似した機能をもつ回路に対して共通のクロックツリーを設け，各クロックツリー内，異なるクロックツリー間で，その下にぶら下がるフリップフロップに見えるクロックの遅延が一様になるようにバッファ挿入を行う(図C)．これをCTS(Clock Tree Synthesis)という．

MPUが命令をデコードした時点で，その命令が駆動するクロックツリーを検出し，必要のないクロックツリーへのクロック供給を根元から止めてしまう．このようにして低消費電力を実現する方式をゲーテッドCTSという．この手法は，ある程度大域的にクロックを止めるので，通常のゲーテッドクロックよりも効果が大きい．一般的に，マスククロックとかゲーテッドクロックという場合は，このゲーテッドCTSのことを指す．

ゲーテッドCTSでは，クロック供給はハードウェアによって，自動的にクロック供給が行われる．しかし，大量のクロックマスク用ゲートでの遅延を揃えるために，高機能なツールが必要である．このため，最近までやりたくてもできない技術だった．大体，昔はクロックラインにロジックを挿入するなど非常識なことと考えられていた．

ゲーテッドCTSがかなわなかった昔，長い間使われない回路へのクロック供給を止めるという中間解が考案された．これは，外部回路でI/Oなどの状態を監視し，アイドル期間がある程度続くようなら割り込みを入力して，割り込みハンドラ内でソフトウェアによる電力制御(クロックを分周したり停止したりする)を行う手法である．



図C　ゲーテッドCTS

(4) その他の技術

MPUの内部回路はすべてが最高の周波数で動作する必要はない．クロックドメイン（一つのクロックが管理する領域）ごとに，処理の性質に応じて最適な周波数を選択することで，消費電力を最適化できる．たとえば，パイプラインを駆動するクロックは最高周波数であることが要求されるが，周辺回路に供給するクロックはそれほど高くなくてよい．とくに，割り込みのサンプリングは，かなり遅い周波数でも実用になる．

また，究極的な方法として，使用していないユニットの電源供給を停止する方法がある．これに関しては，対象となるユニットの再起動に時間がかかるのが欠点である．MPU内の電流逆流防止などの考慮も必要である．

● 動作電圧と動作クロックの同時制御が主流

近年，低消費電力を売りにするMPUは，動作電圧と動作クロックの両方をプログラムの負荷によって最適な値に設定できるようになっている．

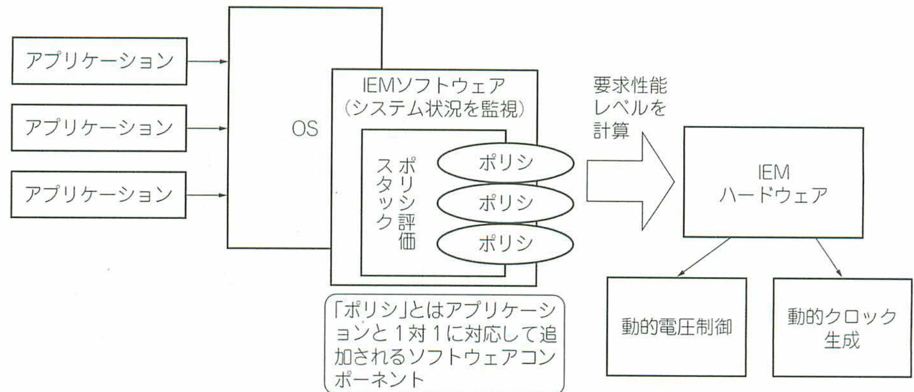
動作クロックの変更は，先に説明したように，比較的簡単に実現できるので昔から行われている[注1]．しかし，動作電圧の変更は容易ではない．MPUとしては動作電圧を変更できるタイミングを通知することしかできない．実際に動作電圧を変更するのは外部回路の役割である．この場合，多段階の電圧に対応したレギュレータ回路が必要になる．

また，MPU自体も動作電圧を変更できるように設計していなければならない．通常のMPUは動作電圧が一意に定められている（変動誤差は±10％程度）．しかし，電源電圧を可変にするには，ある程度の範囲をもった動作電圧を許可しなければならない．たとえば，1.0V～1.6Vの場合，変動誤差があるので，実際には0.9V～1.7V程度の範囲で正常動作が保証されてなければならない．2003年の時点で，このように広範囲の動作電圧に対応したMPUはIntelのXScaleとIBMのPowerPC405LPだけといわれている（ほかにもあるとは思うが）．実際には，次に示すARM11や，今後発表される低電力MPUは，そのような電圧範囲を考慮したものになってくると思われる．

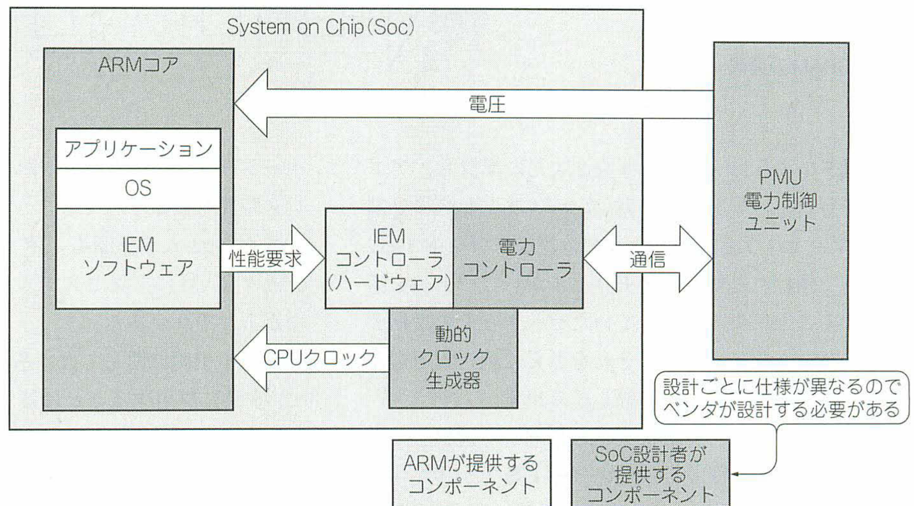
● ARMのIEM（Intelligent Energy Manager）

ARM社の提供するIEM技術は，動作電圧と動作周波数を同時に制御する技術である．基本的な手法は，OSにIEMソフトを追加して負荷状況を監視して，どの程度まで電圧，周波数を下げられるか予想することで最適な電圧と周波数を決定する．この場合，やみくもに電圧や周波数を下げるのではなく，アプリケーションの品質を落とさない（画面が乱れるようなことをしない）程度を見きわめられるのが特徴である．具体的には，アプリケーションのアプリごとにポリシと呼ばれるシステムの動作状況の監視プログラムを提供し，最適な動作を決定する（図D）．

注1：とある低消費電力プロセッサのシンポジウムで，各社の説明員にMotorolaの特許をどのように回避しているのか聞いてみた．しかし，誰もそのような特許の存在を知らなかったのが驚きである．IBMの説明員は，そんな特許があったとしても，Motorolaとはクロスライセンスを結んでいるので心配ないと豪語（？）していた．



図D　IEMの概要



図E
IEMのシステム構成

図Eに IEM 使用時のシステム構成について示す．電圧制御(Power Management Unit：PMU)は外付け，周波数変更(動的クロックジェネレータ)はSoCに内蔵となる．実際の実装では，ARMコアの周辺回路として電圧クランプ回路とレベルシフタが必要である．

IEMを使えば，ARMのSoCの電力を75％削減可能であり，電池寿命を25％延ばせる．ARM社が公表しているムービープレーヤの例では，IEMなしで実行した場合に対し，10fps，15fps，20fpsの画質に対し，それぞれ，45％，37％，32％の電力が削減できる．これはCPUコアをアイドル状態にするだけでは得られない電力効率だそうだ．

簡易版として，ソフトウェアのみで制御するIEM-Oneが利用可能である．最大限に電力を削減するためには，ソフトウェアとハードウェアの両方で省電力を実現するIEM-Twoを提供する．これらはARMコアとは別個のライセンスとなる．

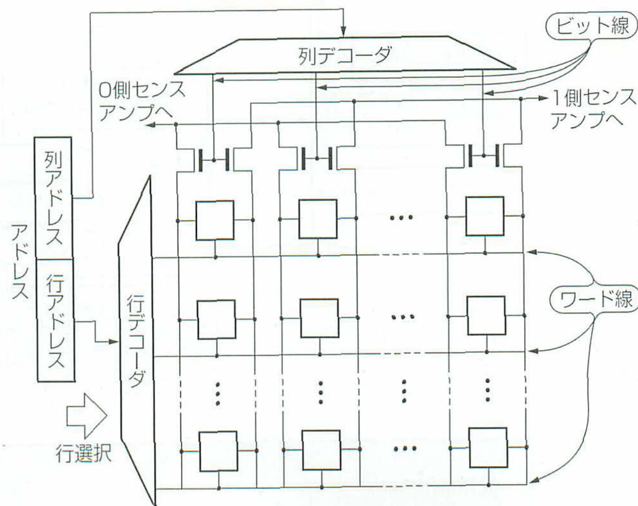
● キャッシュの電力制御

現在のRISCプロセッサはキャッシュの存在が必須である．しかし，MPUの内部でキャッシュがもっとも電力を消費する部分である．消費電力の半分がキャッシュによるものといっても過言ではない．このため，完全低消費電力を実現するためには，ゲーテッドクロックなどで，キャッシュ以外の部分の低消費電力化を行うだけでは不十分で，キャッシュそのものの電力を削減することが重要になる．

(1) ブロック分割

これは，キャッシュメモリを複数のバンク(ブロック)に分割して，必要なブロックのみを活性化してアクセスする方法である．具体的には，各メモリバンクは選択信号をもち，キャッシュをアクセスするアドレスの一部分をデコードすることにより，活性化するバンクを選択する．キャッシュの電力制御の方法として，ごく普通に行われている．

キャッシュメモリはSRAMで構成される．SRAM



図F　SRAMの構成

は，**図F**のようにメモリセルを2次元に配置し，アドレスを行(ロウ)アドレスと列(カラム)アドレスに分割して,アクセスするメモリセルの位置をデコードする．行選択で選ばれた複数のメモリセルが，ワード線を活性化することにより記憶していたデータとその反転データをビット線に出力し，それをさらに列選択で選んだものをセンスアンプで増幅し読み出す．

ブロック分割は，行選択をさらに細かく選択して行うことである．つまり，一部のワード線しか活性化しない．メモリセルはフリップフロップに入出力用のゲートを付加したものであり，このゲートはワード線を活性化して駆動する．駆動するメモリセルが少ないほど消費される電流が少ないのは明らかで，結果として低消費電力を実現できる．

また，キャッシュのタグRAMとデータRAMを独立してアクセス可能にし，タグチェックでヒットしたデータRAMのみを活性化すると，さらに消費電力を低下させることができる．しかし，タグRAMの値を見ないとデータRAMにアクセスできないので，キャッシュのアクセスタイミングを苦しくして速度低下につながるおそれがある．

(2) ウェイ予測

これは，nウェイセットアソシアティブキャッシュにおいて，ヒット/ミスの判別時にすべてのウェイの内容を同時に読み出すのではなく，予測した順番でウェイを読み出す方法である．キャッシュを同時に読み出さない分，電力が少なくなる．この場合，予測を正しく行わないと，キャッシュアクセスに時間がかかってしまうが，スーパースカラ構成を採る場合は，リザベーションステーション(早い話が命令キュー)でそのロス時間も供給されてしまうといわれている．

また，ウェイ予測も，タグRAMの値を見ないとデータRAMにアクセスできないので，速度低下につながるおそれがある．

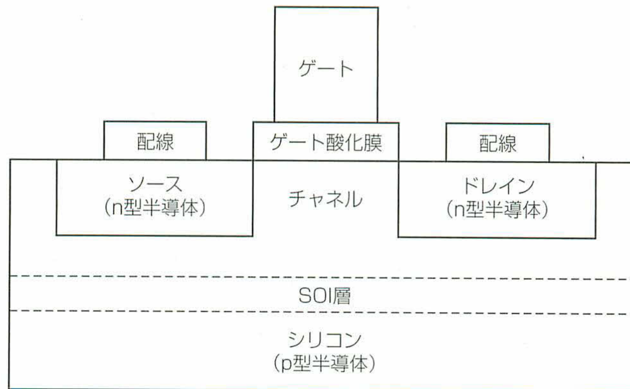
ウェイ予測に関しては，基本特許が多く出ているので,安易に採用することは難しいと思われる．しかし，堂々とウェイ予測を謳うMPUが発表されている現状を考えると，抜け道はいくらでもあるのだろう．

(3) キャッシュの階層化

キャッシュは，通常，L1キャッシュ，L2キャッシュ，L3キャッシュと階層化して使用される．これは，キャッシュのアクセス時間と内蔵できるキャッシュサイズとの兼ね合いで分割されていることが多い．つまり，L1キャッシュ，L2キャッシュ，L3キャッシュの順にキャッシュサイズが増加していくのが普通である．そして，L1キャッシュはL2キャッシュの内容の一部を，L2キャッシュはL3キャッシュの内容の一部をキャッシングする(ビクティムキャッシュはそうなってないが)．

このような階層構造は消費電力を下げる効果もある．一般的に，アクセスするキャッシュのサイズが大きいほど電力を消費するので，時間的，空間的に，できるだけ少ないサイズのキャッシュへのアクセスを優先することで低消費電力化を実現できる.この方式は，特殊な形態のブロック分割といえるかもしれない．

アドレス変換に使用するTLBでは，その内容をキャッシングしたマイクロTLBを最初にアクセスすることがあるが，これも同じ発想である．



図G　nMOSトランジスタ

● トランジスタレベルの低消費電力化

この分野は筆者の専門ではないので簡単に説明する．

トランジスタの集積度はムーアの法則にしたがって増大してきた．1971年に登場した4004と最新のPentium4を比較すると，動作周波数は約2万倍に，トランジスタ数は約2万4000倍にまで増大した．しかし，動作電圧は低下しているが，動作周波数とトランジスタ数の増加とともに消費電力も増加している．この消費電力がMPUを発熱させる要因となっている．Intelの試算では，MPUの発熱は，2007年には核反応炉と同レベルになり，2010年を過ぎた頃にはロケットの噴射口に匹敵し，ほどなく太陽の表面温度に等しくなるという．その意味でも，トランジスタの低消費電力化は必須である．

(1) リーク電流

半導体製造プロセスの微細化が進むと，当然，トランジスタは小さくなる．トランジスタは，シリコンの上に，シリコンと逆極性のソース領域とドレイン領域を形成し，それをゲートでつなぐ(図G)．ゲートに電圧をかけるとソースとドレインの間に電流が流れ，これがスイッチのように働く．

しかし，微細プロセスでは，ゲート長が短くなり，トランジスタが導通し始める電圧(しきい値：V_{th})が低下して，ゲートに電圧をかけなくても漏れ電流(リーク電流)が流れてしまう．このリーク電流(オフリーク電流，または，チャネルリーク電流)のせいで，トランジスタが動いてなくても，電力を消費してしまう．また，これでは，トランジスタがONなのかOFFなのか判別できないので，正常動作のためには，リーク電流を超える電流を流す必要がある．これも消費電力が増大する原因となる．

あるいは，ゲート長の縮小化にともない，ゲート酸化膜(絶縁膜)もほぼ比例して薄くする必要がある．しかし，膜厚が薄くなることにより，トンネル効果で，絶縁膜を通して流れるリーク電流(ゲートリーク電流)が無視できなくなる．

オフリーク電流とゲートリーク電流は，消費電力を増大させる要因になる．これは，電池駆動の機器においてはとくに深刻な問題であり，リーク電流を削減するための手法がいろいろ考案されている．

リーク電流を低減させる抜本的な方法を以下に示すが，とりあえず行われるのが，微細プロセスになるほど動作電圧を下げるという方法である．トランジスタを駆動する電圧が低ければ，流れる電流も小さく，消費電力が下がるという理論である(消費電力は電圧の2乗に比例する)．

(2) オフリーク電流対策

オフリーク電流対策としては，ソースやドレインを形成するシリコン層の厚さを薄くする方法がある．シリコン層にソースやドレインを形成すると，その影響による空乏層(depression region)[注2]の広がりがシリコンの厚みに対して小さく，空乏化されてないシリコンがゲートの下に残る．このシリコンがフローティングボディ(どこにも接続されていない電極)となり，その電位がトランジスタ特性に影響を与える．シリコン層を薄くしてゲートの下のシリコンを完全に空乏化するとフローティングボディがなくなり，また電流が通過できる範囲が狭められる(抵抗が増える)ため，オフ

注2：電荷を運ぶキャリア(電子または正孔)が存在していない領域のこと．空乏があると電流の流れが悪くなる．

リーク電流を低減できる．しかし，シリコン層の厚みはV_{th}に影響を与えるので，これを薄くするとV_{th}が下げられる反面，トランジスタの制御が難しくなる．したがって，やみくもにシリコン層を薄くすればいいというものでもない．

Intelは完全空乏型のシリコン層を使用することで，部分空乏型よりもオフリーク電流を1/100にできると発表している．このトランジスタは完全空乏型基板トランジスタ(Depleted Substrate Transistor)，略してDSTと呼ばれる．DSTでは，V_{th}を下げられるので，トランジスタの動作電圧を低減したり，トランジスタのON/OFF動作を高速化したりするのに役立つ．DSTでは，部分空乏型 SOIトランジスタとは異なり，フローティングボディ効果が発生しないので，従来のトランジスタと同じ方法で回路設計が行えるのが利点であるとして，IBMを牽制している．

(3) ゲートリーク電流対策

ゲートリーク電流対策としては，ゲート酸化膜の素材を変更することが考えられている．ゲート酸化膜とは，ソースとドレイン間を流れる電流をゲートに流れ込まないように絶縁するための絶縁膜で，この部分が薄ければ薄いほどトランジスタは高速に動作する．しかし，ゲート酸化膜が薄くなるとゲートリーク電流が増加する．そこで，ゲート酸化膜を厚くする方法が考えられるが，できることならある程度の薄さも維持したい．

これを解決するため，ゲート酸化膜に誘電率の高い(high-k)材料を用いることで，誘電率の比率で酸化膜を厚くすることが可能になり，リーク電流が減る．これは，高速化技術のために提案されているlow-k材料および銅配線とは対極的であるが，こちらは配線に関する技術である．配線に関しては，寄生容量が電流の流れを妨げ，電力を消費する．このために，層間膜は誘電率の低い材質(low-k)や電気的抵抗の少ない銅配線が好まれるのである．

ゲート酸化膜に何が最適な素材であるかは，半導体メーカーが力を入れて研究している分野であり，現状でこれといった決定版はない．現在，ゲート酸化膜には二酸化シリコン(SiO_2)を使うことが多い．high-kの絶縁膜の目的は，物理的には厚い膜で，薄いSiO_2と同じ電気特性(ゲート容量)を実現することでる．しかしIntelは，将来的には，誘電率が二酸化シリコンの約5倍の酸化ジルコニウム(ジルコニア：ZrO_2)，酸化ハフニウム(HfO_2)，二酸化チタン(TiO_2)，五酸化タンタル(Ta_2O_5)などを使用することを提案している．AMDは誘電率が約2倍の窒化シリコン(Si_3N_4)を使用することを提案している．これらを使用することで，二酸化シリコンと同じ電気的性能を発揮しながら，ゲートリーク電流をその1/10,000～1/1,000に抑えることができるといわれている．

ただ，high-kの絶縁膜は，二酸化シリコンとは違って製造が難しいうえ，シリコンとの相性が悪く，信頼性に問題があるといわれている．また，high-kの絶縁膜では，SiO_2に比べるとキャリア(電子や正孔)の移動度が落ちる(つまり，動作速度が落ちる)という問題もある．これらを解決するのが今後の課題である．新しいゲート絶縁膜が主流になる時期は，早くてもゲート長が65nmの世代だろうといわれている．

High-k絶縁膜の問題を解決するための一つの手段がメタル(金属)ゲートである．従来のゲート電極はポリシリコン(多結晶Si)を使用していたが，NMOSトランジスタやPMOSトランジスタに最適な金属素材を使用してゲートを作る．メタルゲートは抵抗が小さく，ゲート空乏化が発生しないので，トランジスタ性能は劣化しない．

メタルゲートの材質には今後も研究の余地が残されているが，NMOSではTaSiN，PMOSにはTiNが使われることが多いようである．

2002年12月に開催されたIEDM(International Electron Device Meeting)では，high-kの絶縁膜としてHfO_2が一般的になった感がある．これの膜質の改良や製造性の改善のためにNやAl，Siを添加する論文が多数提出されている．

High-k酸化膜とは別の考え方はバックゲートバイアスである．トランジスタのゲートに逆方向(バックゲート)の電圧をかけるとゲートを流れる電流が抑制される．このバイアス電圧をうまく調整すればゲートリーク電流を理論的にはゼロにできる．この技術はIBMやIntelが積極的に研究している．

TransmetaはEfficeonに搭載するLongRun2でトランジスタのV_{th}をソフトウェアで動的に制御することでゲートリーク電流を削減すると発表している．高速化のためにはV_{th}を低くする必要があるが，リーク電流が増加する．とくに90nmプロセスではV_{th}が低くなるため，リーク電流の削減は重要である．逆にV_{th}を上げるとリーク電流は少なくなるが高速動作は期待できない．LongRun2はCMS(Code Morphing Software)でプログラムの負荷を監視し，負荷が少な

い場合はV_{th}を上げる技術である．既存のLongRunを使った場合，Efficeonのゲートリーク電流は144mWであるが，LongRun2では2mWに下げられるという．

LongRun2を実現する具体的な手法は明らかにされてないが，Transmetaによると「回路にほんのわずかな改良を加えた」ということだ．しかし，大方の見方としてはバックゲートバイアスの変形ではないかと考えられている．バックゲートバイアス電圧によってV_{th}を相対的に制御できる．

なお，2003年10月のMicroprocessor Forumで公表されたLongRun2技術は「全体像の10％にすぎない」という話であり，さらに画期的な電力低下技術が隠されている可能性がある．

(4) SOI技術

あるいは，SOI(Sillicon On Insulator：絶縁体の上のシリコン)という技術で，ソース-ドレインとシリコン基板の間に二酸化シリコンによる極薄の絶縁層を組み込む方法がある．SOI層を加えることで，シリコン基板とソースあるいはドレインの間の寄生容量を低減することにより，ソース-ドレイン間の抵抗を減少させ，電流の流れを20～30％よくする技術である．同じ性能(電流の量)であれば，電源電圧を下げることにより消費電力をほぼ半減できる．まあ，これは，低消費電力というよりも高速化技術である．

SOIという技術は，30年以上前から半導体メーカーが研究をしている．その中でも有名なのは，Harris Semiconductor社のSOS(シリコンオンサファイヤ)である．しかし，これは非常に高価なため実用にならなかった．現在は，二酸化シリコンを使用して比較的安価に作られているが，今後も新たな材質の発見が望まれる．

まとめ

今後，ますます重要な技術になると考えられる低消費電力技術について述べてきた．その根本原理は，低い動作周波数と低い動作電圧を実現すること，回路の内部状態をできるだけ変化させないこと，一度に駆動する回路を減らすことである．これらを組み合わせることで，今後もいろいろな制御方式が考案されていくことと思われる．トランジスタレベルでの低消費電力化については，現在発展途上というところだろうか．また，トランジスタレベルでの低消費電力化技術は高速化技術と組で研究されているので，高速化技術の章も参照してほしい．

参考文献

1) 中川靖,「64ビットRISCマイクロプロセッサ　V_R4131」,『NEC Device Technology』, 2001, No.74

第6章

外的要因と内的要因，ハードウェア割り込みとソフトウェア割り込みの違いを理解する

割り込みと例外の概念とその違い

割り込みには，MPUの動作とはまったく非同期に外部のデバイスが要求するハードウェア割り込みと，プログラム中に明示的に分岐命令を記述するソフトウェア割り込みがある．また，プログラムの実行結果によって発生する予期しない事象を例外と呼ぶ．ハードウェア割り込みは外的要因で発生するが，ソフトウェア割り込みと例外はMPUの内的要因で発生する．例外と割り込みの区別はそれぞれのMPUアーキテクチャ上の決め事であり，その本質は同じと考えられる．

15年ほど前，筆者は割り込みというものの概念がよくわからなかった．

MPUは与えられた処理を順次こなしていく．その処理に割り込んで，いったい何をするのか．処理Aをこなしながら処理Bも行う必要があるのなら，AとBを同時に実行するようにプログラムすればよいではないか．

例外についても同様に，行っていることは固定アドレスに分岐して戻ってくることである．それはサブルーチンコールと何が違うのか．

以降の解説は，15年を経て筆者が感じ取った割り込みと例外の意義やしくみである．

1 MPUにおける割り込みと例外

● 割り込みとは何か

割り込みとは，一連の仕事をしているときにその仕事を中断させて別の仕事をさせることである．割り込みされる側からすると，予期しないタイミングで発生するのが特徴である．

MPUでアプリケーションプログラムを実行する場合，通常は割り込みを意識しない．割り込みが発生するとそれまでの処理は中断され，特定の割り込み処理を行って元の処理に復帰する．アプリケーションプログラム側は割り込まれたことに気付かない(**図1**)．

MPUのプログラム実行順序としては，図のように一筆書き状態の順番でプログラムを実行しているにすぎないが，人間の時間感覚で見ると，本来の処理と割り込み処理が平行に実行されたように見える．本来のプログラムが気付かないうちに並行動作が行われる……ここに割り込みの本質がある．

● ハードウェア割り込みとソフトウェア割り込み

割り込みは大きく分けて，MPUに接続された外部のデバイスが要求する**ハードウェア割り込み**と，プログラムで明示的に要求する**ソフトウェア割り込み**の二つがある．

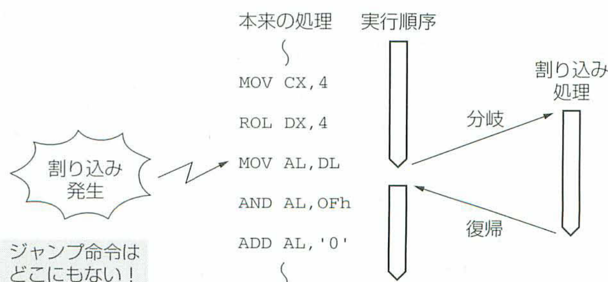


図1
割り込み処理の概念
(ハードウェア割り込み)



図2　ソフトウェア割り込み

ハードウェア割り込みとは，**図1**のように外部からの要因でジャンプ命令などを使わずにプログラムの実行が分岐することである．ハードウェア割り込みはアプリケーションプログラムには見えない．外部のハードウェアの状態が変わったことを検出し，それにしたがって処理が必要な場合に利用する．一般的に，外部割り込みはMPUの処理とは非同期に行われる．

一方，ソフトウェア割り込みは，割り込み処理へ切り替える命令をアプリケーションプログラム中に明示的に記述する(**図2**)．この意味で，ソフトウェア割り込みはサブルーチンコールのようにもみえる．たいていのMPUには，ソフトウェア割り込みを発生させるためのトラップ命令やシステムコール命令が用意されている．

● 例外とは何か

一般的に割り込みは，プログラムの実行とは無関係(非同期)に発生するが，プログラムの実行結果によって発生する予期しない事象がある．たとえば，ゼロ除算，オーバフロー，アドレスエラー，ページフォルト(TLBミス)などである．これらの発生によってもプログラムの処理は中断され，それら予期しない状態を処理するプログラムが実行される(**図3**)．

これらは，要因がプログラムの実行そのものにあり，外部からの要因によって割り込まれたわけではないので，とくに**例外**と呼ぶ．

「例外」を辞書で引くと，「通例の原則にあてはまらないこと．一般の原則の適用を受けないこと．また，そのもの．」とある．コンピュータの世界でもイメージは同じだが，命令の処理が通常と同じようには終了しない事象を表す．

どのような事象が発生したときに例外となるのかは，MPUのアーキテクチャによって異なる．たとえば，定義されていない命令コードを実行すると，あるアーキテクチャでは例外となるが，あるアーキテクチャではNOPと同じ動作となり，そのままプログラムを実行し続ける．

● 割り込みと例外の区別

要因発生後の動作，つまり割り込み処理へ分岐する動作は，割り込みも例外も共通である．しかし，割り込みの場合は元のプログラムに復帰するのが前提であるが，例外は場合によっては，致命的な事象と判断してプログラム処理を中止(アボート)することもある．

事象発生後の挙動が同じという点で，割り込みと例外は言葉のうえでの区別のようにも思える．実際，割り込みと例外を同一視するアーキテクチャのMPUも多い．そのような場合，外的要因によるハードウェア

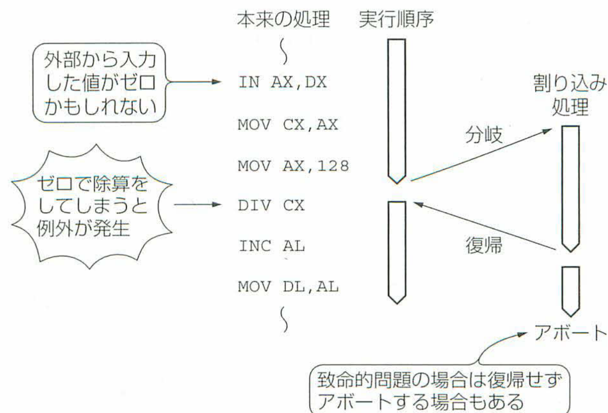


図3　例外の概念

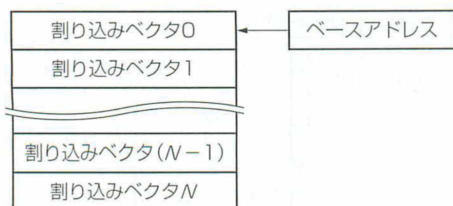


図4　割り込みベクタテーブル

割り込みを**外部割り込み**，内的要因による例外とソフトウェア割り込みを**内部割り込み**と呼んで区別する．

割り込みと呼ぶか例外と呼ぶかは，そのMPUのアーキテクチャ上の決め事である．ここでは原則として，外的要因によるものを割り込み，内的要因によるものを例外として話を進める(とはいえ，「ソフトウェア例外」とはあまり呼ばないが…)．

● ベクタとハンドラ

割り込みが発生すると割り込み処理へ分岐するわけだが，どこに分岐するかを示すものを**割り込みベクタ**と呼ぶ．そして，割り込み処理ルーチンのことを，**割り込みハンドラ**と呼ぶ．また，割り込みと呼ぶか例外と呼ぶかに対応して，ベクタとハンドラも，割り込みベクタ，割り込みハンドラ，例外ベクタ，例外ハンドラと呼ばれる．

● 割り込みの受け付け，NMIとリセット

割り込みとは，本来の処理の途中で別の処理を行わせることだが，処理の内容によっては，実際に連続して実行しないと意味をなさない場合や，途中で割り込み処理が実行されては都合の悪い場合もある．そのような場合は，割り込みの受け付けを禁止することもできる．

しかし，外的要因の中には非常に緊急性を有する事象もある．もしそれが発生した場合は，割り込まれると都合の悪い処理であっても，その緊急の割り込み処理を実行する必要があるだろう．このような重要な割り込みは，割り込み受け付け禁止ができない割り込みとして**ノンマスカブル割り込み**(Non Maskable Interrupt，NMI)を使う．通常，割り込みと呼ぶ場合は，ソフトウェアで割り込みの受け付けを禁止することができる**マスカブル割り込み**のことをいう．

MPUのアーキテクチャによっては，リセットも割り込みや例外に分類するものがある．割り込みベクタがプログラマブルなMPUであっても，さすがにリセット時は特定のアドレスから実行を開始したり，特定アドレスのメモリを読み込み，その値をアドレスとして実行を開始する(リセットベクタ)．

また，ノンマスカブル割り込みという意味では，リセットもノンマスカブルな割り込みといえる．しかもNMIよりも優先度が高く，MPUの中ではもっとも優先度の高い割り込みといえる．

● 割り込みベクタテーブル

CISC系MPUの多くは，割り込みや例外に対する割り込みベクタの値，つまり割り込みハンドラのアドレスを自由に設定することができる．その割り込みベクタをある決められた順序でメモリ上に並べたものを**割り込みベクタテーブル**と呼ぶ．

多くの場合，割り込みベクタテーブルのベースアドレス，つまり先頭の割り込みベクタが格納されているアドレスは物理アドレスの0番地である．MMUをサポートするMPUでは，この割り込みベクタのベースアドレス(物理アドレスで指定する)を変更可能な場合が多い．そのため，割り込みベクタのベースアドレスを保持する特別なレジスタが用意されている．このベースアドレスレジスタの値を変更することで，割り込みベクタテーブルを任意のアドレスに配置することができる(**図4**)．

一方，RISC系MPUの多くは，割り込みベクタの値がアーキテクチャで一意に決められているので，割り込みベクタテーブルというものは存在しないことが多い．さらに，割り込みベクタの値は仮想アドレスだが，対応する物理アドレスは1対1で決まっている(たとえば，アドレス変換されない)ことが多い．

2 外部割り込みと例外の動作の概要

ここでは，ハードウェア(外部)割り込みと例外の動作について解説する．以降ではとくに明記しない限り，ハードウェア割り込みを単に「割り込み」と示すことにする．ソフトウェア割り込みについては，コラム1を参照してほしい．

● 割り込まれるプログラムの影響

割り込みや例外は，割り込まれるプログラム側からすれば，意図しない場所で秘密裏に処理される．このときの動作はどうなっているのだろうか．まず，プログラムの実行を規定する要因を考えよう．ある瞬間のプログラムを完全に再現するには，

- プログラムの命令コードとデータ
- プログラムでアクセス可能なすべてのレジスタの値
- PC(プログラムカウンタ)の値

●SR(ステータスレジスタ)の値

といったデータが一意に定まっていればよい．これらの情報をコンテキストと呼ぶ．ここでPCは，いうまでもなく，現在実行している命令コードのアドレスである．SRは，PSW(Program Status Word)やPSR(Program Status Register)とも呼ばれ，条件分岐用の条件フラグや実行レベルが含まれる(x86でいうところのFLAGレジスタ)．

これらのうち，プログラムの命令コードとデータは，そのプログラムの実行が終了するまで物理メモリまたは補助記憶上に存在しているので，とくに気にする必要はない．レジスタの値は壊されると困るので，割り込みハンドラでは，そこで使用するレジスタの値をスタックなどに退避しておき，例外ハンドラを抜けるときに退避しておいた値を書き戻してやればよい．レジスタは割り込みハンドラで使用しないこともあるが，PCとSRの値は必ず変更される．つまり，PCとSRがプログラムの挙動を性格付ける．

結論として，各レジスタやPCとSRを割り込み処理の前に保存し，割り込み処理を終了した後で元に戻してやれば，割り込まれたプログラムは何も知らずに処理を継続することができる．

● 割り込み/例外発生時の動作

実際に，割り込みや例外が発生したときのMPU内の動きについて見てみよう(**図5**)．多くのMPUでは，割り込みや例外が発生すると，PCとSRを自動的に特定の場所に退避するようになっている．また，外部から割り込みアクノリッジ(ベクタ)を読み込むMPUもある(詳細は，実際のMPUでの動作の項目で説明)．

CISC系MPUでは，割り込みや例外が発生するとPCとSRを(割り込み用)スタックに退避し，割り込みからの復帰を指示する命令(RETIなど)を実行すると，スタックからPCとSRの元の値を取り出して，新たにPCとSRに設定し直す．

RISC系MPUでは，スタックアクセス(＝メモリアクセス)を行うと処理速度が低下してしまうので，退避専用の特殊レジスタに値を格納する．割り込みハンドラの終了を指示する命令は，PCとSRの値をこの特殊レジスタから取り出す．これらのレジスタは1組しか用意されていないのが普通で，多重に割り込みや例

Column 1　ソフトウェア割り込みとサブルーチンコール

ソフトウェア割り込みは，トラップ命令やシステムコール命令などのように，プログラムで明示的に記述して積極的に発生させる割り込みである．ソフトウェア割り込みは，OSが提供するサービスを得るためのシステムコールのインターフェースとして利用される．意味的にはサブルーチンコールと大差はない．それではなぜ，ソフトウェア割り込みというわずらわしい(わけでもないが)手順を踏むのであろうか．それには少なくとも二つの理由がある．

一つは実行レベルの問題である．WindowsやLinuxでは，ユーザープログラムはMPUの提供するユーザーモードで実行されている．それに対して，OS内部はカーネルモードで実行される．通常のサブルーチンコールでは現在の実行レベルを保持するので，ユーザープログラムからコールしたサブルーチンでは特権命令を実行できない．ソフトウェア割り込みによって，実行レベルを特権レベルに上げることができる．

二つ目はコールするアドレスの問題である．WindowsやLinux上のユーザープログラムは，基本的にすべて仮想アドレス上で動作する．一方，OSのサービスルーチンの先頭アドレスは一意に決まっている．その先頭アドレスを明示的にユーザープログラムで指定するには，仮想アドレスがどの物理アドレスに変換されるのかを知る手段がない以上，一般には不可能である．割り込みベクタテーブルは，通常，システムに一つだけ存在するので，OSのサービスを割り込みハンドラで指定するようにすれば，すべてのタスクから同じOSのサービスルーチンを実行できてむだがない．

歴史的にながめれば，保護やアドレス変換がない昔のMPUでは，システムコールがサブルーチンコールによって行われていた．これは仕方のないことである(というか，それ以外の方法はなかった)．しかし，比較的新しいところでは，OS/2でもシステムコールをサブルーチンコールで実現していた．その当時，すでにMS-DOSではシステムコールにINT命令を使用していたので，OS/2は先祖返りといえなくもない．なぜ，そのようなしくみを採用したのか，IBMの見解を聞いてみたいものである．OS/2を動作させるMPUが，アドレス変換がまだ洗練されてなかった80286だったことが一因かもしれない．

おもしろいところでは，Windows CEや一部のLinuxのシステムでは，システムコールにアドレスエラーを利用している．MPUにはトラップ命令やシステムコール命令が用意されているのに，なぜこうなっているのかは謎である．

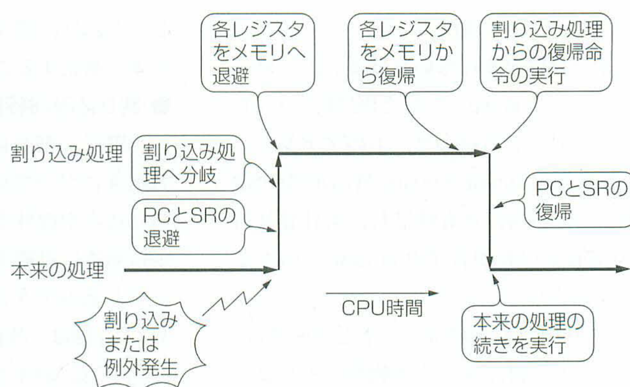


図5　割り込み/例外処理の動作の概要

外が発生すると値が上書きされてしまう．多重に割り込みが発生する可能性がある場合は，スタックなりメモリなりに内容を退避する必要がある(RISCにもスタックという概念はある)．

割り込み発生前と割り込みハンドラからの復帰後で，プログラムが使用しているレジスタの値は保存されなければならない．このレジスタの退避/回復処理は大量のメモリアクセスを伴うので，性能低下につながる．それを避けるため，アーキテクチャによっては割り込みハンドラのみがアクセスできる，通常のレジスタとは独立したレジスタを提供していることもある．このような構造を**レジスタバンク**と呼ぶ．ARMなどのアーキテクチャは，例外の種類ごとに数種類のレジスタバンクを備える．

また，割り込みからの復帰命令はMPUによって異なるが，だいたい次のような名称で呼ばれる．

RETI　(RETurn from Interrupt)
RETE　(RETurn form Exception)
IRET　(Interrupt RETurn)
ERET　(Exception RETurn)

この名称によって，そのMPUのアーキテクチャが割り込み/例外のことを，割り込み(Interrupt)と呼んでいるか，例外(Exception)と呼んでいるかを知ることができる．

● 割り込み発生と割り込みマスク

一般的なMPUでは，一度に一つの割り込み要求しか受け付けないようにするため，割り込み発生時には新たな割り込みの受け付けができなくなる．ソフトウェアによる割り込みや例外処理中に発生する割り込みは，割り込み処理が終了するまで待たされる．具体的には，復帰命令を実行して割り込みが許可されるまで，新たな割り込みは受け付けない．

一方，ソフトウェアによる割り込みや例外処理中に発生する例外に関しては，禁止(マスク)する手段がない．多くの場合はその例外処理に移行するが，発生する例外の種類によっては2重例外による致命的例外となり，MPUの実行が停止する場合もある．

一般に，例外処理中には割り込みの受け付けが禁止されるが，意図的にSRを書き換えれば割り込みの受け付けを可能にすることもできる(多重割り込みについては後述)．

● 割り込み許可とマスク

通常，割り込みには許可ビットとマスクビットが用意されている．許可ビットとは，割り込みの受け付けを許可するか否かを指定するビットである．割り込み発生時に新たな割り込みを受け付けないようにする機構は，この許可ビットを自動的に受け付け禁止に設定することで実現されていることが多い．

一方，マスクビットとは，割り込みをマスク(覆い隠す＝禁止する)ためのビットである．MPUが割り込み端子を1本しかサポートしていない場合は，マスクビットの意味はない．許可ビットとまったく同じ意味となるからである．

後述するように，複数の割り込み入力がある場合，それぞれの割り込み要求に対して1対1にマスクビットが存在し，割り込み要因ごとに独立して割り込みを禁止する場合にマスクビットを使う．そして，MPUとして全割り込みの受け付けを許可するか否かを許可ビットで指定する．いずれにせよ，許可ビットとマスクビットの両方で割り込みが許可されていないと，割り込み要求は受け付けられない(**図6**)．

● 複数割り込みと優先順位

割り込み要求はたいていの場合，MPUの外部端子によって通知される．バスサイクルで与えられる

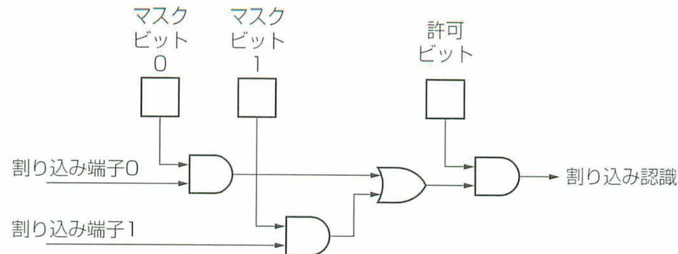


図6　許可ビットとマスクビット

MPUもあるが，ごく稀なケースなのでここでは割愛する．

MPUによっては，外部割り込み入力が1本という場合もあるが，実際にシステムを構築する場合には，割り込み要因が10を越えることは珍しくなく，複数の外部割り込みを扱う要求が出てくる．割り込みが複数ある場合は，割り込みの優先順位をどうするかも問題である．

MPUによっては，外部割り込みを優先順位付きのレベルで識別できる．この場合，割り込み端子は複数本からなり，その端子状態が割り込み要求のレベルを表す．たとえば，割り込み端子の本数が3本なら，0～7の8種類のレベルを要求できる．このレベルはそのまま割り込みの優先順位となり，MPU内に記憶されている基準レベルと比較され，それより優先順位が高い場合は割り込みを受け付ける．

図7は，要求レベルの値が大きいほど優先順位が高いものと仮定し，0の場合が割り込みなしの状態となっているときの割り込みを認識するしくみである．この基準レベルはソフトウェアで任意に変更できる．つまり，ある優先順位の割り込みを処理している場合は，それより優先順位の低い割り込み要求を受け付けないようにもできる．この場合，基準レベルが割り込みのマスクとして機能している．逆に，現在より優先順位の高い割り込み要求が発生すると受け付けてしまう．それを防ぐためには，ソフトウェアで現在処理中の優先順位を最高位に上げておかなければならない．

● 割り込みコントローラ

図7のようなMPUでは，割り込み要因に対応した値(レベル)を割り込み入力端子に入力する必要がある．しかし，外部割り込みを発生させる一般的な外部デバイスは，割り込み要求時に割り込み出力端子をアサートする機能しかもたず，それ自身ではレベルを生成することができないものが多い．

そのような場合は，プライオリティエンコーダ(優先順位の符号化器)を使用し，割り込み要因に対応したレベルをMPUに入力できるようにする．また，複数の割り込みが同時に発生した場合は，もっとも優先順位の高い割り込み要因のレベルをMPUに入力する(**図8**)．このように，複数の割り込み要因を優先順位を考慮してMPUに伝達するデバイスを，**割り込みコントローラ**と呼ぶ．

MPUに割り込み端子が複数あっても，レベル入力方式でない場合もある．その場合は，各割り込み端子自体が優先順位をもっている．たとえば，INT0，INT1，INT2という割り込み端子があれば，INT0＜INT1＜

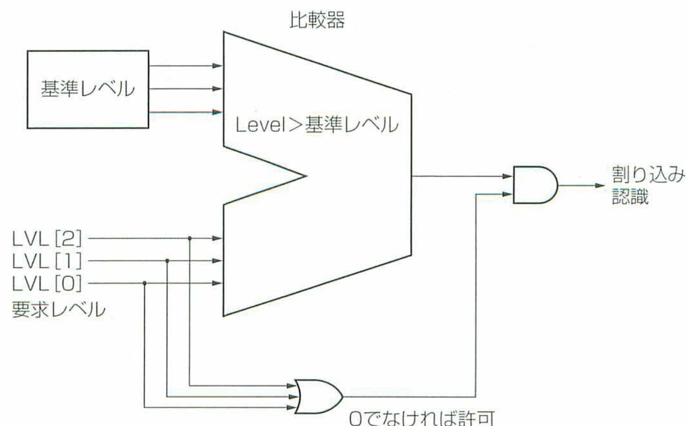


図7　レベル方式による優先順位付き複数割り込み入力

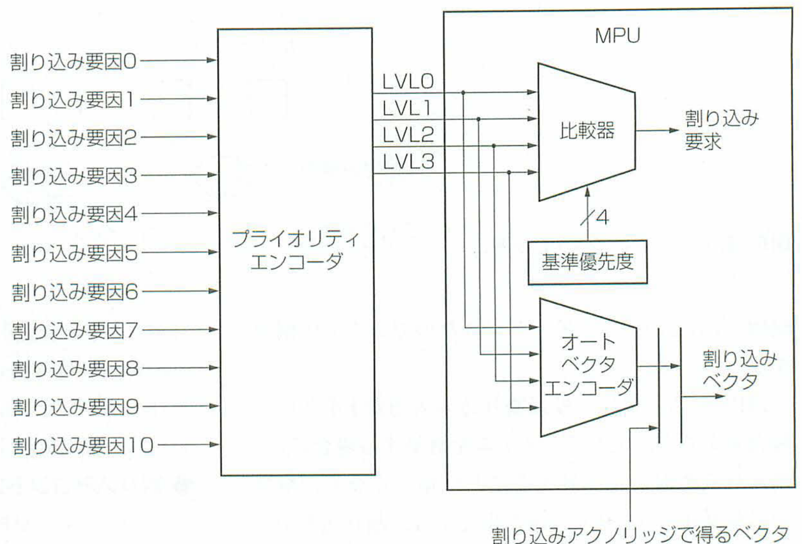


図8　レベル方式による割り込み入力本数の拡張

INT2の順に優先順位が高く，複数の割り込み端子が同時にアサートされる場合は，より高い優先順位の割り込みが受け付けられる．MPUに用意されている割り込み入力本数では足りない場合は，外部に割り込みコントローラをカスケード接続するなどして割り込み入力を拡張する必要がある(**図9**)．

また，MPUによっては，複数の割り込み端子を有していても，それらにハードウェア的な優先順位がないこともある．その場合，マスカブル割り込みの割り込みベクタは1種類で，あとはソフトウェアで「よきに計らえ」ということになる．

具体的には，すべての割り込み端子の状態がソフトウェアから見えるようになっていて，それを見ながらソフトウェアで適当に優先順位をつけて処理することになる．この場合，割り込みを認識するソフトウェアのステップ数が増加するので割り込みハンドラの処理が重くなる．しかし，ハードウェア構成が単純なので，RISC系のMPUではこの構成がしばしば採用される．

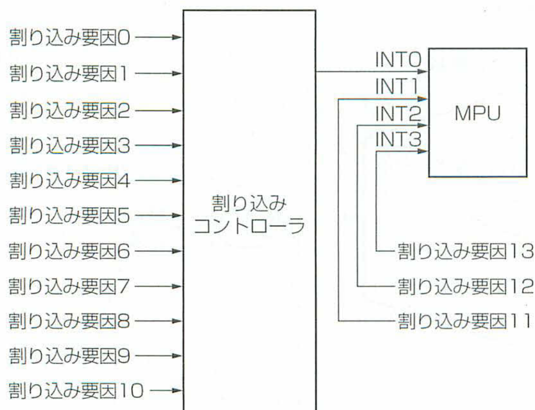


図9　割り込み入力端子に優先順位がある場合の割り込み入力の拡張

● 多重割り込み

複数の割り込み要因が優先順位付きでMPUに入力される場合，優先順位の低い割り込み処理中に，より優先順位の高い割り込みが発生する可能性がある．

通常，割り込み処理中(割り込みハンドラの実行中)は新たな割り込みの受け付けはマスクされる．しかし，割り込みハンドラ内でも，割り込み許可ビットをセットして，より優先度の高い割り込み要求の受け付けを許すようにもできる．これにより，より優先度が高い割り込みが発生した場合，そちらの割り込み処理を開始することができる．これを**多重割り込み**と呼ぶ．

多重割り込みは，CISC系MPUなどのPCやSRがスタックに保存されるMPUなどでは，とくに考慮が必要となるような問題はない．しかし，RISC系MPUなど，PCとSRが専用レジスタに退避されるだけの方式では，多重割り込みを許可する前に，その専用レジスタの内容が書き潰されないように，元の値をスタックなどの領域に退避しておく必要がある．

● 割り込みを受け付けるタイミング

割り込み要求が発生したとき，MPUがその要求を受け付けるタイミングはいつだろうか．それは，MPUが割り込み処理を行うのに都合のよいタイミングである．

いくらなんでも，命令を実行している途中(具体的

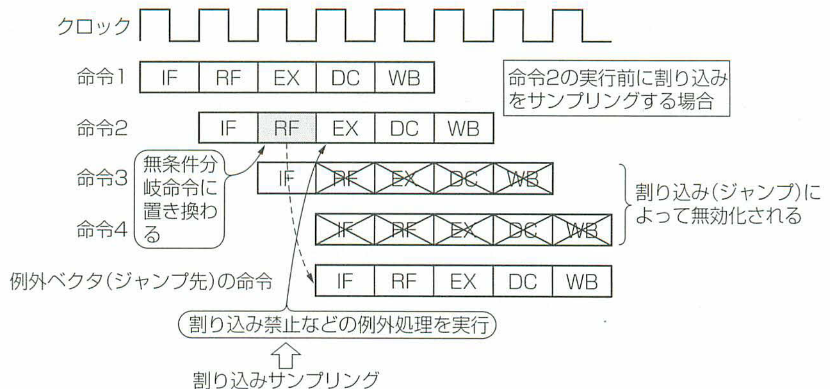


図10　割り込み機能の実現

には結果をデスティネーションレジスタにライトバックする前に)で割り込みを受け付けたりしたら，一時的に保持している値が壊れてしまうので，正しい結果をライトバックできない．この意味からも，割り込みは命令の実行終了後，次の命令の実行前のタイミングで受け付けられるのが普通である．

RISCでは，1命令の実行時間は基本的に1クロックなので，たいていの場合は，割り込みを要求してから1クロック後には割り込みが受け付けられる．ただし，FPUの除算命令などは実行に50クロック以上もかかることもあり，その場合には割り込みを受け付けるまでに最悪50クロック程度かかることになる．

CISCの場合，1命令で行う処理の複雑さゆえ，命令の実行時間は通常1クロック以上である．たとえば，文字列転送命令や，倍精度の浮動小数点命令の実行には200クロック以上かかることも珍しくない．

これでは割り込み応答性のよいリアルタイムOSを作ることはさすがに難しい．そこで，CISCのMPUでは，実行時間の長い命令に関しては，例外的に命令実行の途中で割り込みを受け付けるようになっている．

割り込み発生時にスタックに積まれるPCの値は，一般には，次の命令のPC(Next PC)であるが，命令実行中に割り込みを受け付ける場合は，実行を中断した命令(実行中の命令)のPC(Current PC)である．このため，割り込みハンドラでRETIなどの復帰命令を実行すると，中断した命令から実行が再開される．MPU内部では，命令の再開処理がうまくいくようなしくみが用意されているのである．

● 割り込み機能の実装

実際に，MPUで割り込み機能を実装するためにはどうするのだろうか．簡単に説明すると，命令がパイプラインを流れる間に割り込みを受け付けると，その命令を割り込みベクタへ分岐するジャンプ命令に置き換える．

割り込みと例外はほとんど同じ処理になるので，割り込みがあるかないかを調べるタイミング(サンプリングという)は，例外検出を行うタイミングと同じ場合が多い．つまり，例外発生時も，その命令を例外ベクタへのジャンプ命令に置き換えることで実現できる．

たとえば，IF(命令フェッチ)，RF(デコード)，EX(実行)，DC(データアクセス)，WB(ライトバック)からなる5段パイプラインのMPUを考える．例外として考えられるのは，RFステージでの未定義命令例外(ブレークポイントやシステムコールを含む)とDCステージでのアドレスエラー(データのTLBミスを含む)やEXステージの結果に依存するトラップやオーバフローなどである．

このような場合，RFステージかDCステージで割り込みをサンプリングするのが普通である．割り込み応答を良くしたい場合は，RFステージとDCステージの両方で割り込みのサンプリングを行う．

ただし，DCステージで割り込みをサンプリングする場合は，命令のデコードはすでに終了しているので，単純に例外ベクタへのジャンプ命令に置き換えることはできない．この場合は，その命令をジャンプ命令に置き換えるというよりは，次にフェッチする命令をジャンプ命令に置き換えると考える．処理的には，RFステージでサンプリングするよりも複雑である(それならば，すべてDCステージでサンプリングすればよいという考えも当然ある)．

図10に，RFステージで割り込みをサンプリングする場合の割り込み処理の概略を示す．パイプラインの制御はジャンプ命令と同じでよいので，割り込みを受け付けた後続命令を無効化する処理もジャンプ命令と

同様の制御で実現できる．

割り込みだけでなく，例外処理も同じ実装でよいが，命令フェッチ時のアドレスエラーやTLBミスの場合は，例外ベクタへのジャンプ命令をフェッチしてくると思えばよい．

とくに例外は，実行（EXステージ）が終わらないと発生の有無がわからない場合もあるので，DCステージでのサンプリングは必須である．DCステージは，演算結果を書き戻す（WBステージ）直前であり，無効な結果を書き戻さないようにするための最後のチャンスである（割り込みなら1命令後で発生してもかまわない）．DCステージで例外を検出した場合は，WBステージでの書き込みを禁止して，次にフェッチする命令をジャンプ命令に置き換える．

3 割り込みと例外処理の実際

それでは，実際のMPUにおける割り込みと例外処理について，いくつかのアーキテクチャのMPUを取り上げて説明する．

● x86の場合

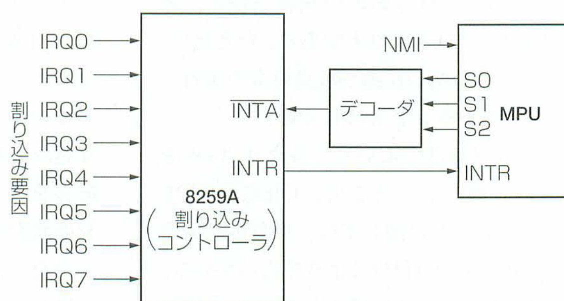
x86での割り込みはハードウェア割り込みのことを指し，周辺デバイスからの割り込み要求によって発生する．例外は，トラップ，フォールト，アボートに区別される．トラップとはINT xといったソフトウェア割り込み，フォールトは主としてMMU関連の例外，アボートは処理が続けられないようなエラー発生時の例外である．

▶ハードウェア構成

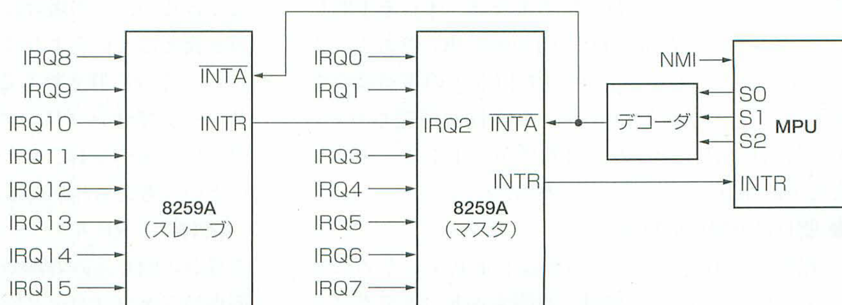
x86アーキテクチャのMPUでは，割り込みコントローラとしてIntelの8259AというLSIを想定している（最近では，APIC = Advanced Programmable Interrupt Controllerがその役割を果たす）．MPUと割り込みコントローラは，**図11**（a）のように接続される．8259Aは1個で八つまでの割り込み要因しか処理できない．それ以上の割り込み要因が必要な場合は，**図11**（b）のように割り込みコントローラをカスケード接続して対応する．

割り込み発生時の割り込みコントローラの動作を**図12**に示す．これを割り込みアクノリッジサイクルと呼ぶ．8259Aは，割り込みを出力するデバイスからの割り込み要求を察知すると，MPUの外部端子である割り込み要求端子（INTR）をアサートし，外部割り込み要求の存在を知らせる．

MPUは外部割り込みの存在を感知すると，割り込みアクノリッジを示す信号（S2～S0端子）を出力する．そこで，割り込みコントローラはデータバスに割り込み番号（ベクタ）を与えて割り込みの種類を示す．割



（a）1個のみ接続する場合



（b）カスケード接続の場合

図11 x86用割り込みコントローラ8259の接続

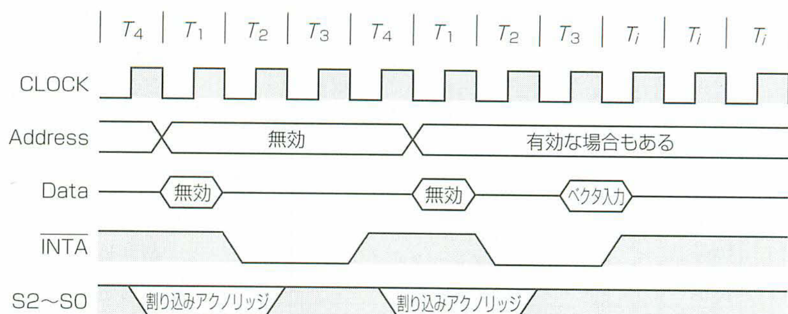


図12　x86の割り込みアクノリッジサイクルの動作

り込みアクノリッジ・サイクルが2回発生するのは8259Aのつごうである．

1回目で割り込みが発生したことを認識し，2回目で割り込みベクタを返す．なお，ここでいうベクタとは，割り込みハンドラの先頭アドレスではない点に注意してほしい．

割り込みアクノリッジサイクル自体は，「要求された割り込みを受け付けた」という意味ももっている．割り込みアクノリッジが発生しないということは，要求された割り込みが無視されたということである．これは，割り込みがマスク(禁止)されている場合に起こりえる．

その場合，割り込みコントローラは割り込み要求端子をアサートし続け，割り込みアクノリッジが発生するのを待つのが普通である．通常，割り込み要求端子は，割り込みアクノリッジが発生するまでアサートし続ける．

▶ **割り込み番号とその要因**

x86がサポートする割り込み番号とその要因を**表1**に示す．ソフトウェア割り込みを発生するINT命令は，パラメータとして0～255の割り込み番号を指定することができる．このため，INT命令によってす

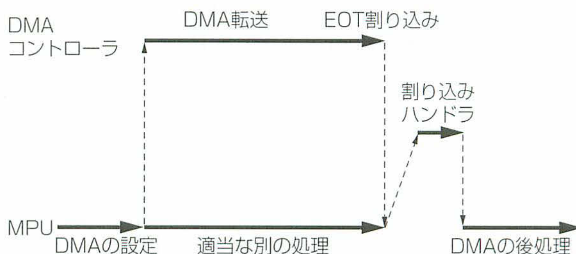
Column 2　割り込みとポーリング

割り込みの利点の一つとして，ある処理の終了を割り込みで通知するようにしておけばその間に別の処理を並行して実行できることが挙げられる．

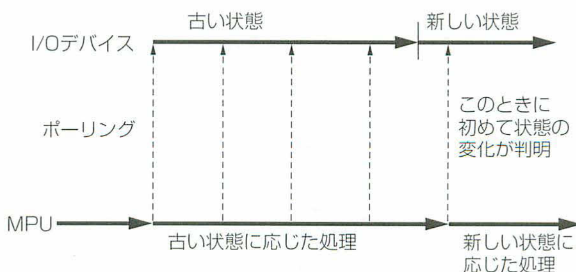
たとえば，DMAの待ち合わせに割り込みが多用される．DMAコントローラの多くは，転送終了時にEOT(End Of Transfer)またはTC(Terminal Count)といった割り込みを発生する．DMA転送を割り込みで待ち合わせる処理のイメージを**図A**に示す．

このように，動作の終了で割り込みを出力する機能をもたないデバイスによる処理の待ち合わせには，そのデバイス内のステータスを定期的にチェックして，処理が終了したかどうかを判定しなければならない(**図B**)．このように定期的にステータスの状態をチェックすることを**ポーリング**という．

ポーリングは，ソフトウェアによる単純なループであることも少なくない．ポーリングによる処理の待ち合わせは，状態の変化を知るまでに遅れが生じるので，割り込みと比べると効率が悪い．また，その間に別の処理を行えないという点でもポーリングの効率は悪い．



図A　DMA転送終了割り込みによる処理の待ち合わせ



図B　ポーリングによる処理の待ち合わせ

表1　x86の割り込み番号とその要因

割り込み番号	種　類	要　因
0x00	フォールト	除算エラー
0x01	フォールト	デバッガ割り込み（トレース）
0x02	アボート	NMI
0x03	トラップ	INT 3（ブレークポイント）
0x04	トラップ	INTO
0x05	フォールト	配列境界違反
0x06	フォールト	無効命令
0x07	フォールト	コプロセッサ無効
0x08	アボート	ダブルフォールト
0x09	アボート	コプロセッサセグメントオーバラン
0x0A	フォールト	無効TSS
0x0B	フォールト	セグメント不在
0x0C	フォールト	スタック例外
0x0D	フォールト	一般保護例外
0x0E	フォールト	ページフォールト
0x10	フォールト	コプロセッサエラー
0x11	フォールト	アラインメントチェック
0x12	アボート	マシンチェック
0x13	フォールト	ストリーミングSIMD拡張
0x12～0x1F		予約済み（使用不可）
0x20～0xFF		ユーザー用（外部割り込み/INT命令）

べての割り込み/例外を発生させることが（理論上）可能である．外部割り込みの割り込み番号は，先ほど説明した割り込みコントローラから与えられる．

▶ リアルモードでの動作

割り込み/例外処理の挙動は，リアルモードとプロテクトモードで異なる．

リアルモードでは，0x00000番地から始まる256エントリの割り込みベクタテーブルで，割り込み/例外の割り込み番号とその処理ハンドラのアドレスが対応付けられる．割り込みベクタテーブルの各エントリは，2バイトのオフセットアドレスと2バイトのセグメントアドレスから構成される．

割り込み/例外が発生すると，MPUはフラグレジスタ，CSレジスタ，IPレジスタをスタックにプッシュして例外スタックフレームを作り，発生した割り込み/例外の割り込み番号に対応する割り込みベクタテーブルのエントリからオフセットアドレスとセグメントアドレスを読み出す．そして，それぞれの値をIPレジスタ，CSレジスタに設定することにより，処理ハンドラに分岐する．

▶ プロテクトモードでの動作

プロテクトモードの場合は，割り込みベクタテーブルではなく，割り込みディスクリプタテーブル（IDT）が使用される．割り込みディスクリプタテーブルの先頭アドレスは，0x00000番地に固定ではなく，IDTRレジスタによって設定される．

割り込みディスクリプタテーブルは，割り込み番号とその処理ハンドラのアドレスを決定するゲートディスクリプタとを対応付ける256エントリのテーブルである．ゲートディスクリプタは2バイトのセレクタ値，4バイトのオフセットアドレス，1バイトのスタックコピーカウント，1バイトのゲートの種類から構成される8バイトのデータである．

大雑把にいえば，リアルモードでの割り込みベクタテーブルのエントリに対して，オフセットアドレスが2バイトから4バイトに拡張されたと思えばよい．そして，セレクタ値が間接的にセグメントの先頭アドレスを指し示す．

プロテクトモードにおいて割り込み/例外が発生すると，スタックポインタが特権レベル0のスタックポインタに切り替わる．そして，その新しいスタックに古いスタックポインタ（SS：ESP）をプッシュし，その後，EFLAGSレジスタとCSレジスタとEIPレジスタの値をプッシュして，ゲートディスクリプタで指定された処理ハンドラに分岐する（**図13**）．割り込み/例外処理を行った後，IRET命令を実行すると，特権レベル0スタックからSS：ESP，EFLAGS，CS：EIPを回復する．

x86における割り込みと例外の差異は，処理ハンドラに分岐した時点で，新しいFLAGSレジスタまたはEFLAGSレジスタの割り込み許可ビットが禁止（割り込み発生時）になっているか，前の値を引き継いでいる（例外時）かだけである．

● MC680x0の場合

▶ ハードウェア構成

68000系では，割り込みコントローラを含む周辺デバイスとして，MC68901というMFP（Multi-Function Peripheral）が存在する．とくに，組み込み制御用途のMPUでは専用の周辺デバイスが用意され，割り込みコントローラもそれに含まれていることが多い．割り込みコントローラは，各社独自のASICとして供給されることもある．

図14に，680x0での割り込みアクノリッジサイクルを示す．680x0での割り込みのベクタ番号は一定し

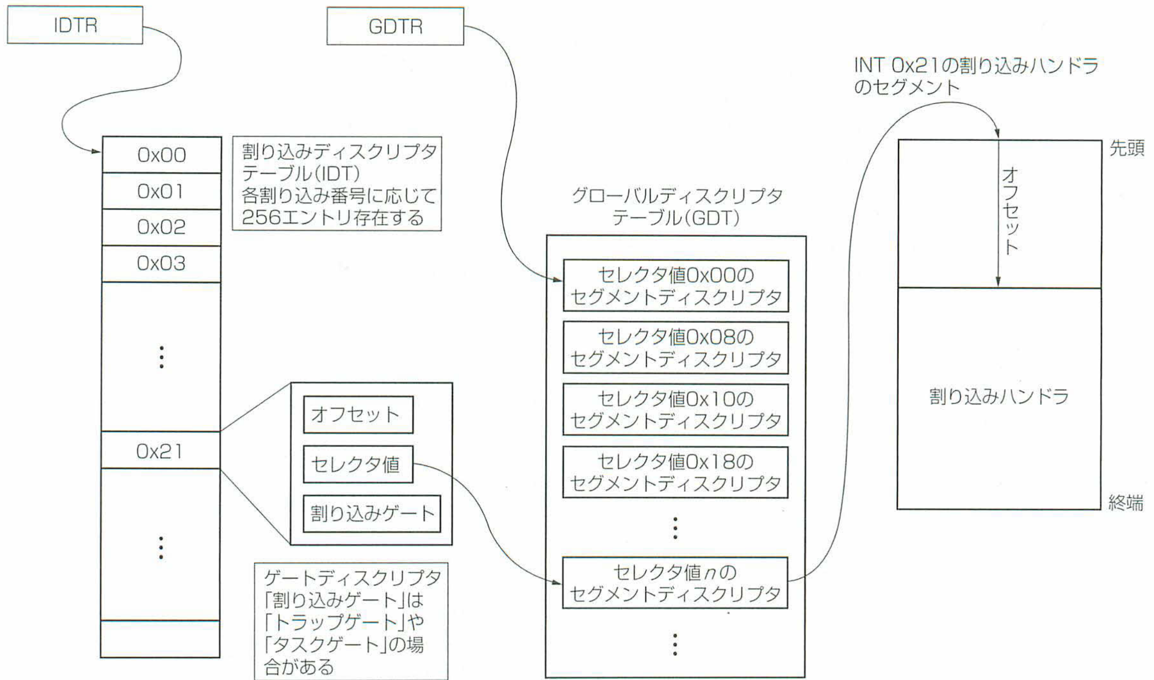


図13　割り込みハンドラの選択(プロテクトモード)

ておらず，MPU外部の割り込みコントローラによって与えられる．割り込みを受け付けると，MPUは割り込みアクノリッジバスサイクルを発行して，割り込みコントローラにベクタ番号を問い合わせる．割り込みコントローラは，発生している割り込みの種類に応じてベクタ番号(64～255)を返すか，オートベクタを使用する(AVEC端子をアサートする)かを決定する．

オートベクタというのは，割り込みの優先レベル(1～7)に固定のベクタ番号である．具体的には，優先レベルに24を加えた25～31がベクタ番号となる．オートベクタは，システムと密接した割り込み処理に利用されることが多いようである．

もし，割り込みアクノリッジバスサイクルに対して何も応答が返らない場合はスプリアス割り込みとなる．これは割り込みの要因が不明な割り込みで，MPUとしては処理する方法がわからない．通常のシステムでは，ノイズによる誤動作などとして，スプリアス割り込みは無視される(割り込みハンドラはRTEのみ)．

▶ 割り込み/例外の動作

MC680x0の割り込み/例外処理は，例外ベクタテーブルと例外スタックフレームを使用する．ベクタベースレジスタ(VBR)は，256個の例外ベクタからなる1024バイトの例外ベクタテーブルの先頭アドレスを保持する．例外ベクタは，リセットベクタを除いた例外処理ルーチンの先頭アドレスである．

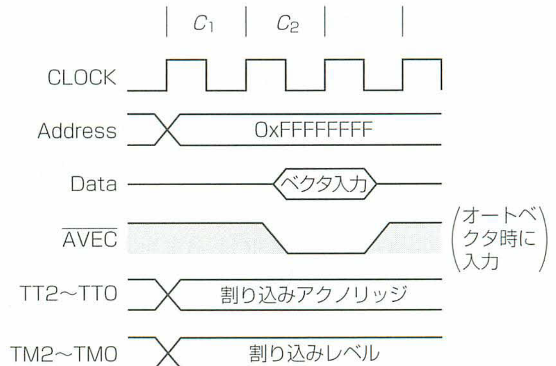


図14　680x0の割り込みアクノリッジサイクル

表2に，例外ベクタテーブルの内容を示す．このうち，リセットベクタはISP(割り込みスタックポインタ)の初期値とPCの初期値(実行開始アドレス)からなる．例外ベクタの格納されているアドレスは，例外の種類に応じてMPUが自動的に割り当てる8ビットのベクタ番号から決定される．また，いくつかの例外については，外部デバイスが例外ベクタを供給する．例外ベクタアドレスは，例外ベクタを4倍し，VBRの値に加算して決定される．

割り込み処理はスーパバイザスタックに例外から復

表2　680x0の例外ベクタテーブル

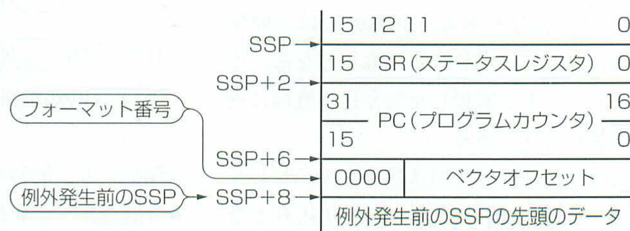
ベクタ番号	オフセット	割り当て
0	0x000	リセット時割り込みスタックポインタ
1	0x004	リセット時プログラムカウンタ
2	0x008	アクセスフォールト
3	0x00C	アドレスエラー
4	0x010	不正命令
5	0x014	整数ゼロ除算
6	0x018	CHK，CHK2命令
7	0x01C	FTRAPcc，TRAPcc，TRAPV命令
8	0x020	特権違反
9	0x024	トレース
10	0x028	ライン1010エミュレータ(未実装Aライン命令コード)
11	0x02C	ライン1111エミュレータ(未実装Fライン命令コード)
12	0x030	(予約)
13	0x034	コプロセッサプロトコル違反
14	0x038	フォーマットエラー
15	0x03C	未初期化割り込み
16～23	0x040～0x05C	(予約)
24	0x060	スプリアス割り込み
25	0x064	レベル1割り込みオートベクタ
26	0x068	レベル2割り込みオートベクタ
27	0x06C	レベル3割り込みオートベクタ

ベクタ番号	オフセット	割り当て
28	0x070	レベル4割り込みオートベクタ
29	0x074	レベル5割り込みオートベクタ
30	0x078	レベル6割り込みオートベクタ
31	0x07C	レベル7割り込みオートベクタ
32～47	0x080～0x0BC	TRAP #0～#15命令
48	0x0C0	FPアンオーダ状態での分岐またはセット
49	0x0C4	FP精度落ち
50	0x0C8	FPゼロ除算
51	0x0CC	FPアンダフロー
52	0x0D0	FPオペランドエラー
53	0x0D4	FPオーバフロー
54	0x0D8	FPシグナリングNot a Number
55	0x0DC	FP未実装データ形式
56	0x0E0	MMU構成エラー
57	0x0E4	MC68851で使用
58	0x0E8	MC68851で使用
59～63	0x0EC～0x0FC	(予約)
64～255	0x100～0x3FC	ユーザー定義ベクタ

帰するための情報を積む．これらは，例外の種類によって異なる，例外スタックフレームと呼ばれる構造を採る．例外スタックフレームは，SR(ステータスレジスタ)，PC(プログラムカウンタ)，ベクタのオフセット，スタックフレームの形式を示す領域と，追加情報からなる．

例外/割り込み処理の後，RTE命令を実行すると例外スタックフレームからMPUの再実行に必要な情報が読み込まれて，実行を再開する．

MC680x0で定義されている例外スタックフレーム(フォーマット0)を**図15**に示す．例外スタックフレームの種類は，MC68000，MC68010，……，MC68060と世代が進むごとに(対処的に？)拡張され，最終的には10種類を超えた．付け焼き刃のようで，アーキテ



例外の種類	PCが指す位置
割り込み	次の命令
フォーマットエラー	RTE命令またはFRESTORE命令
TRAP #N	次の命令
不当命令	不当命令
Aライン命令	Aライン命令
Fライン命令	Fライン命令
特権違反	特権違反を発生させた命令の最初のワード
浮動小数点命令実行前	浮動小数点命令
未実装整数	未実装整数命令
未実装実効アドレス	未実装実効アドレスを使用した命令

図15
680x0の例外スタックフレームの構造
(フォーマット0)

モード						
ユーザーモード	特権モード					
		例外モード				
ユーザー	システム	スーパバイザ	アボート	未定義	IRQ	FIQ

◆ 汎用レジスタ

ユーザー	システム	スーパバイザ	アボート	未定義	IRQ	FIQ
R0						
R1						
R2						
R3						
R4						
R5						
R6						
R7						
R8						R8
R9						R9
R10						R10
R11						R11
R12						R12
R13(SP)		R13	R13	R13	R13	R13
R14(LR)		R14	R14	R14	R14	R14
R15(PC)						

◆ ステータスレジスタ

ユーザー	システム	スーパバイザ	アボート	未定義	IRQ	FIQ
CPSR						
		SPSR	SPSR	SPSR	SPSR	SPSR

図16 ARMのレジスタ構成

クチャとしてはあまり美しくない．

● ARMの場合

▶ レジスタ構成

ARMのアーキテクチャでは，割り込み/例外発生時に，ユーザーレジスタの退避の必要性をなくすため，レジスタバンクが用意されている．このレジスタバンクは，割り込み/例外の種類（モード）に応じて5バンクが独立して存在する．このレジスタバンクのレジスタの一部はユーザーモードのレジスタと共通になっていて，モード間で共通にアクセスできる（**図16**）．

多くのモードではR13とR14を固有にもっている．R13には，そのモードでのスタックポインタの値が格納され，R14には割り込み/例外からの復帰アドレスが自動的にセットされる．R14には，割り込み/例外を発生した次の命令のアドレスが格納されるので，ユーザーモードへの復帰時には，処理モードに応じて適当な値をR14から減算してPCに格納する．

高速割り込みモード（FIQ）では，コンテキストスイッチのオーバヘッドを軽減するため，R8～R14がモード固有のレジスタとして用意されている．例外スタックフレームは存在しない．その代わり，ステータスレジスタは，新しいレジスタバンクに存在するSPSRに退避される．PCは，新しいレジスタバンクのR14に退避される．

▶ 割り込み/例外の動作

割り込み/例外発生時のMPUの動作は，次のとおりである．なおARMでは，ベクタアドレスは一つについて4バイトの領域しかないので，通常は処理ルーチンへの分岐命令が格納されている．(6)と(7)がソフトウェアによる処理である．

(1) 例外に対応する処理モードに移行

(2) 戻りアドレスを新しい処理モードのレジスタバンクのR14に退避

(3) CPSRの値を新しい処理モードのレジスタバンクのSPSRに退避

(4) CPSRの所定ビットをセットして外部割り込み不可にする

(5) 処理モードに応じた例外ベクタアドレス（**表3**）へ分岐する

(6) 例外処理を実行

(7) ソフトウェア割り込み，未定義命令トラップからの復帰時

➡`MOVS PC, R14`　（R14をPCに格納）

IRQ，FIQ，プリフェッチアボートからの復帰時

表3　ARMの例外ベクタアドレス

割り込み/例外の種類	モード	ベクタアドレス
リセット	SVC(スーパバイザ)	0x00000000
未定義命令	UND(未定義)	0x00000004
ソフトウェア割り込み	SVC(スーパバイザ)	0x00000008
命令フェッチメモリフォールト	Abort(アボート)	0x0000000C
データアクセスメモリフォールト	Abort(アボート)	0x00000010
アドレス例外(26ビットアドレス)	Abort(アボート)	0x00000014
IRQ(通常の割り込み)	IRQ	0x00000018
FIQ(高速割り込み)	FIQ	0x0000001C

➡SUBUS　PC，R14，#4(R14から4を減算してPCに格納)

データアボートからの復帰

➡SUBUS　PC，R14，#8(R14から8を減算してPCに格納)

(命令の最後のSは同時にSPSRをCPSRに回復することを意味する)

なお，多重割り込みを行っている場合は，R14(戻りアドレスの基準)がスタックにある．この場合は多重レジスタ転送命令の

```
LDMIA　R13!，{R0-R3，PC}^
```

によって，例外からの復帰ができる(同時にCPSRを回復する)．R13はスタックポインタであり，作業用レジスタとして使われるR0～R3がスタックに退避されている場合を示している．レジスタリストの終わりの^が，CPSRを同時に回復することを指定する．

▶ 最新アーキテクチャでは割り込み機構を高速化

ARMはv6アーキテクチャで，例外/割り込み処理の高速化を目指している．具体的には，

- 新しい割り込みスタック機構(SRS，RFE命令)
- 命令によるモード変更(CPSIE，CPSID命令)
- 発生順序を規定しないアボートをサポート
- 低レイテンシモードの採用(実装依存)
- ベクタ割り込みモードをサポート

である．

● MIPSの場合

▶ ハードウェア構成

MIPS系のMPUは，通常5本の(マスカブル)割り込み端子をもっているが，それらの間に優先順位はない．すべてソフトウェアでの処理に任されている．また，割り込みを受け付けても割り込みアクノリッジサイクルは発行しない．さらに，割り込みは端子の状態が原因レジスタの特定のフィールドにそのまま見えているだけなので，割り込みを確実に認識するためには，割り込み処理が終了するまで割り込み端子の状態を保持する必要がある．

通常のMPUでは，割り込みアクノリッジサイクルが発行されると割り込み要求を取り下げてよい(その割り込みは受け付けられたことが保証される)．MIPSでは，特定のI/Oポートにアクセスしたら割り込み要求を取り下げるというしくみを，外部回路で実現しなければならない．

▶ 割り込み/例外の動作

MIPSの割り込み例外処理は単純である．ほとんどすべての例外は，共通のベクタアドレスへ分岐する．例外スタックフレームは存在せず，ステータスレジスタは例外ビット(EXLまたはERL)がセットされることで特権レベルに移行したことを示す．

一方，PCは特定の特権レジスタ(EPCまたはErrorEPC)に退避される．割り込み/例外の要因は，ほとんどの場合同じアドレス(共通例外ベクタという)に分岐するので，原因レジスタに格納される例外コードを読み出して区別する．**表4**に，原因レジスタに格納される例外コードを示す．

割り込み/例外発生時のMPUの動作は，次のとおりである．(1)～(3)は，外部割り込みの場合である．例外発生時は，直接(4)に移行する．(7)～(9)がソフトウェアによる処理である．

(1) 割り込み要求が発生(INT0～INT4)

(2) INT0～INT4端子の状態とSRのマスクビット(IM0～IM4)の論理積(AND)が原因レジスタの割り込み保留領域(IP0～IP4)に反映される

(3) IP0～IP4のどれか一つが1であり，かつSRの割り込み許可ビット(IE)が1なら割り込みが発生する

(4) カーネルモード(相当)に移行する(EXLまたはERLが1)．同時に割り込み不可になる(EXLやERLが1のときは割り込み不可)

(5) 戻りアドレスを特定の特権レジスタ(EPCまたはErrorEPC)に退避

表4　MIPS系の原因レジスタの例外コード

例外コード	略　号	説　明
0	Int	割り込み
1	Mod	TLB変更例外
2	TLBL	TLB不一致例外（ロード，命令フェッチ）
3	TLBS	TLB不一致例外（ストア）
4	AdEL	アドレスエラー（ロード，命令フェッチ）
5	AdES	アドレスエラー（ストア）
6	IBE	バスエラー（命令フェッチ）
7	DBE	バスエラー（ロード，ストア）
8	Sys	システムコール
9	Bp	ブレークポイント
10	RI	予約済み命令例外
11	CpU	コプロセッサ使用不可例外
12	Ov	演算オーバフロー例外
13	Tr	トラップ
14	VCEI	命令仮想コヒーレンシ例外
15	FPE	浮動小数点演算例外
16〜22	未使用	
23	WATCH	ウォッチ例外
24〜30	未使用	
31	VCED	データ仮想コヒーレンシ例外

(6) 発生要因に応じた例外ベクタアドレス（**表5**）へ分岐する
(7) 外部割り込みの場合は，割り込みの要因を取り下げる
(8) 割り込み処理を行う
(9) ERET命令を実行する
(10) EXL＝1の場合は，EPCのアドレスに分岐しEXLを0にする．ERL＝1の場合はErrorEPCのアドレスに分岐しERLを0にする

例外ベクタアドレスは，リセット直後のステータスレジスタのBEVビットをクリアするまでと，BEVビットをクリアした後で異なる．BEVとはBootstrap Exception Vectorの略で，まだ，キャッシュやTLBを初期化する前の状態を表す．ソフトウェアではそれらの初期化後にBEVビットを0にクリアすることが要請されている．このため，BEVが1の間は非キャッシュで非マップ（アドレス変換されない）領域が例外ベクタになっている．

▶ 最新アーキテクチャでは割り込み機構を高速化

MIPSの割り込み方式は単純でわかりやすいが，その反面，高速な処理には適さない．そこでMIPS社は，2001年10月に発表した拡張機能で，割り込み処理を高速化する機構を強化した．詳細は不明だが，ARMと同様なレジスタバンクを16組もち，割り込みの種類に応じて16種の割り込みベクタを生成するアーキテクチャになるという．

表5　MIPS系の例外ベクタアドレス

例外・割り込みの種類	アドレス
リセット，NMI	0xFFFFFFFFBFC00000
キャッシュエラー	0xFFFFFFFFA0000100 (BEV=0) 0xFFFFFFFFBFC00300 (BEV=1)
TLB不一致（ミス）(EXL=0)	0xFFFFFFFF80000000 (BEV=0) 0xFFFFFFFFBFC00200 (BEV=1)
XTLB不一致（ミス）(EXL=0)	0xFFFFFFFF80000080 (BEV=0) 0xFFFFFFFFBFC00280 (BEV=1)
その他	0xFFFFFFFF80000180 (BEV=0) 0xFFFFFFFFBFC00380 (BEV=1)

● SH(SuperH)の場合

▶ ハードウェア構成

SHの割り込みは，4ビットの優先順位（レベル）方式を採用している．

SH-1/SH-2では，8本の外部割り込み端子（IRQ0〜IRQ7）と内蔵する周辺ユニットからの割り込みがMPUへの割り込み要因となる．これらの割り込み要因は，5本の割り込み優先順位レジスタ（IPR）で独立して優先順位を指定することができる．いずれかの割り込みが要求されると，それに対応した優先順位がMPUに入力される．

一方，SH-3では，6本の外部割り込み端子（IRQ0〜IRQ5），16本のポート割り込み（PINT0〜PINT15），内蔵周辺ユニットからの割り込みに優先順位を与える方式のほか，4ビットの優先順位（IRL0〜IRL3）を直接外部から入力することもできる．SH-4では，4ビットの優先順位入力（IRL0〜IRL3）がユーザーに直接見えるようになっている．

いずれにしろ，割り込み要求（優先順位入力）が，ステータスレジスタ（SR）内の割り込みマスク領域（IMASK）の値よりも優先度が高いときに割り込みを受け付ける．割り込みマスク領域の初期値は最高の優先順位になっているので，MPUの初期化段階で適当な値をIMASKに設定する必要がある．

▶ 例外ベクタの構成

SHの例外ベクタの構成は，SH-2までとSH-3以降ではまったく異なっている．SH-1/SH-2は例外要因それぞれに対して，0x00000000番地から始まる例外ベクタテーブルのオフセットが規定されている（**表6**）．この方式は，MC680x0の方式に酷似している．

〔表6〕SH-1/SH-2の例外ベクタ

例外要因		ベクタ番号	ベクタテーブル
パワーオンリセット	PC	0	0x00000000
	SP	1	0x00000004
マニュアルリセット	PC	2	0x00000008
	SP	3	0x0000000C
一般不当命令		4	0x00000010
(システム予約)		5	0x00000014
スロット不当命令		6	0x00000018
(システム予約)		7	0x0000001C
		8	0x00000020
CPUアドレスエラー		9	0x00000024
DMAアドレスエラー		10	0x00000028
割り込み	NMI	11	0x0000002C
	USER BREAK	12	0x00000030
(システム予約)		13	0x00000034
		:	:
		31	0x0000007C
トラップ命令(ユーザーベクタ)		32	0x00000080
		:	:
		63	0x000000FC
割り込み	IRQ0	64	0x00000100
	IRQ1	65	0x00000104
	IRQ2	66	0x00000108
	IRQ3	67	0x0000010C
	IRQ4	68	0x00000110
	IRQ5	69	0x00000114
	IRQ6	70	0x00000118
	IRQ7	71	0x0000011C
	内蔵周辺	72	0x00000120
	:	:	:
	内蔵周辺	255	0x000003FC

一方，SH-3以降では例外ベクタテーブルを参照せず，直接共通の例外ベクタ(リセット用と他に3種類)にジャンプする方式に変更された(**表7**)．実際に，どの種類の例外が発生したかはEXPEVT(一般例外用，TLBミスかも？)，INTEVT(割り込み用)，TRA(TRAPAのパラメータの4倍)レジスタに格納されている例外要因の値で区別する．この方式は，MIPSの方式に近い．具体的には，リセットがP2(非キャッシュ，非TLBマップ)領域の0xA0000000に固定されている．割り込みと一般割り込み例外は，ベクタベースレジスタ(VBR)が示すアドレスからのオフセットとなっている．割り込みはVBR + 0x600，TLBミスがVBR + 0x400，一般例外がVBR + 0x100である．

SHにおける割り込みのアーキテクチャは，SH-1からSH-4へとMPUが進化するにつれて簡略化される方向にあるようだ．

▶割り込み/例外の動作

実際の割り込み処理の流れを示す．(1)と(2)は外部割り込みの場合で，例外の場合は直接(3)に移行する．

表7　SH-3/SH-4の例外ベクタ(抜粋)

例外要因		ベクタアドレス	例外要因
パワーオンリセット		0xA0000000	0x000
マニュアルリセット		0xA0000000	0x020
TLB多重ヒット		0xA0000000	0x140
リードアドレスエラー		VBR + 0x100	0x0E0
リードTLBミス		VBR + 0x400	0x040
リードTLB保護違反		VBR + 0x100	0x0A0
ライトアドレスエラー		VBR + 0x100	0x100
ライトTLBミス		VBR + 0x400	0x060
ライトTLB保護違反		VBR + 0x100	0x0C0
一般不当命令例外		VBR + 0x100	0x180
スロット不当命令例外		VBR + 0x100	0x1A0
初期ページ書き込み		VBR + 0x100	0x080
TRAPA命令		VBR + 0x100	0x160
USER BREAK TRAP		VBR + 0x100	0x1E0
NMI		VBR + 0x600	0x1C0
外部割り込み	IRL = 0000	VBR + 0x600	0x200
	IRL = 0001		0x220
	IRL = 0002		0x240
	:		:
	IRL = 1110		0x3C0
内蔵周辺からの割り込み		VBR + 0x600	0x400
			:
			0x760

(7)～(10)がソフトウェアでの処理である．

(1) 割り込み要求が発生(IRL0～IRL3)
(2) SRの割り込みマスク(I0～I3 = IMASK)と比較して優先度が高ければ割り込みが発生する
(3) 例外要因レジスタ(INTEVTなど)に割り込み要因コードがセットされる
(4) SRとPCがSSRとSPCに退避される
(5) SRのブロックビット(BL)，モードビット(MD)，レジスタバンクビット(RB)が1にセットされる
(6) 割り込みハンドラへジャンプする
(7) 多重割り込みを許可する場合は，
- SSR，SPCの値をスタックに退避する
- IMASKを許可する優先順位に設定する
- BLビットを0にする(割り込み許可)

(8) 割り込み処理を行う(BL = 0の場合は，より優先度の高い割り込みを受け付け可能)
(9) 多重割り込みを許可する場合は，
- BLビットを1にする(割り込み禁止)
- スタックからSSR，SPCを回復する

(10) RTE命令を実行する
(11) SSR，SPCがSR，PCにセットされる(割り込まれた元にジャンプする)

なお，SH-4では，割り込みコントローラの設定

表8　PALコード例外エントリ(Alpha21214)

エントリ名	タイプ	オフセット	説　明
DTBM_DOUBLE_3	フォールト	0x100	仮想ページテーブル参照時データTBミス．レベル3フロー使用
DTBM_DOUBLE_4	フォールト	0x180	仮想ページテーブル参照時データTBミス．レベル4フロー使用
FEN	フォールト	0x200	浮動小数点不許可
UNALIGN	フォールト	0x280	非整列データ参照
DTBM_SINGLE	フォールト	0x300	データTBミス
DFAULT	フォールト	0x380	データフォールト，仮想アドレス符号チェックエラー
OPCDEC	フォールト	0x400	不正命令コード，機能フィールド
IACV	フォールト	0x480	命令アクセス違反，仮想アドレス符号チェックエラー
MCHK	割り込み	0x500	マシンチェック
ITB_MISS	フォールト	0x580	命令TBミス
ARITH	同期トラップ	0x600	算術例外，FPCR更新
INTERRUPT	割り込み	0x680	割り込み(ハードウェア，ソフトウェア，AST)
MT_FPCR	同期トラップ	0x700	MT_FPCR命令の発行
RESET/WAKEUP	割り込み	0x780	リセット，スリープモードから起床

(ICRレジスタのIRLMビット)で，4ビットの優先順位入力を独立した4本の割り込み要求として利用することもできる．この場合，IRL0，IRL1，IRL2，IRL3の優先順位は，それぞれレベル13，10，7，4として扱われる．

● Alpha(21264)の場合

▶ ハードウェア構成

Alphaアーキテクチャの例外処理は，他のMPUと毛色が違っている．Alphaでは，アプリケーションプログラムが実行する命令コードの他に，例外などの特権操作を記述するためのPAL(Privileged Architecture Library)コード専用命令を定義している．

PALコード専用命令はMPUごとに固有の命令セットで，同じAlphaプロセッサでも互換性があるとは限らない．これにより，システムのOSやハードウェア構成の違いに応じてPALコードを用意することで，複数のシステムにAlphaアーキテクチャを搭載できる．

しかし，これは製造元のDEC(その後Compaqに買収され消滅)の方便のようにも思える．OSの構造を変更した程度で固有のシステムを構成できるほど単純なものではあるまい(ハードウェア構成にも依存する)．

個人的には，PALコードというのは最小限の特権機能を提供する命令を供給することで，OSの設計と並行してチップの設計を早期に行うためのものだと考える．換言すれば，PALコードとはOS記述用のマイクロコードのようなものであるといえる．PALコードによれば，どんなOSでも記述できる(と思われる)．

▶ 例外ベクタの構成

割り込み/例外処理もPALコード専用命令と通常命令の組み合わせで記述される．この総称をPALコードと呼ぶ．逆に言えば，PALコードの存在こそがAlphaアーキテクチャの最大の特徴である．PALコードが呼び出される要因には，次の5種類がある．

- リセット
- ハードウェア例外(MCHK，ARITH)
- メモリ管理例外
- 割り込み
- CALL_PAL命令

つまり，割り込み/例外事象と，それとは独立して直接PALコードを呼び出すCALL_PAL命令である．PALコードのエントリは，PAL_BASEレジスタからのオフセットで定義される．PAL_BASEの値はリセット後に0となるが，ソフトウェアで変更可能である．

割り込み/例外が発生すると，命令の制御は例外のタイプに応じて定義されているPALコードのエントリに移行する．例外処理からの戻りPCはEXC_ADDRレジスタとリターン予測スタックに格納される．

表8は，Alpha21264のPALコード例外エントリとPAL_BASEレジスタからのオフセットを示す．

PALコードで記述された処理の終わりにHW_RET命令(PAL専用命令)を実行すると，リターン予測スタックからPCを読み出して割り込み/例外から復帰する．

▶ 割り込み/例外の動作

Alphaでは6本のハードウェア割り込みと15本のソフトウェア割り込みが提供されている．ハードウェア割り込みは，IRQ_H[5:0] 端子により独立した6種類の要求ができる．これらの要求は，CM_IERレジスタ

のEIEフィールドで独立してマスクが可能である．

6本の割り込みの間に優先順位はなく，ソフトウェアで優先順位づけして処理する必要がある．ソフトウェア割り込みはSIRRレジスタのSIR[15:1]の各ビットに1を書き込むことで要求する．そして，要求されているハードウェア/ソフトウェア割り込みは，それぞれISUMレジスタのEI[5:0]フィールド，SI[15:1]フィールドに反映されている．

割り込み/例外に関連するレジスタとして，もう一つEXC_SUMレジスタがある．これは例外(トラップ)の契機になった命令の種類を格納するレジスタで，トラップ要因やPALコードが行うべき指示などが含まれている．

EXC_SUMレジスタはトラップ発生時に更新され，例外ハンドラの最初にフェッチされるブロック内でリードする場合のみ有効である．つまり，例外ハンドラの最初のキャッシュブロック内にHW_MFPR命令(内部レジスタをリードするPAL専用命令)を置かなければならない．

● SPARC(Version 8)の場合

▶ハードウェア構成とレジスタウィンドウ

SPARCのアーキテクチャは，ARMと同じくバークレー大学のRISC研究の成果に基づいている．そのアーキテクチャでもっとも特徴的なのは，レジスタウィンドウという構成である．

整数ユニット(IU)は32ビットの汎用レジスタを136個もっている．このうち8個はグローバルに参照できるが，残りは手続きごとに割り当てられ，引き数の授受を高速に行う．これがレジスタウィンドウで，一つのウィンドウは24個のレジスタからなる．

内訳は，R24～R31が手続きの呼び出し元とオーバラップする(引き数用)．R16～R23は，手続き内でローカルに使用できる．R8～R15は，手続きが呼び出す手続きとオーバラップする．手続きの最初にレジスタウィンドウを切り替えることで，レジスタの値を退避させることなく，レジスタを自由に使用できる(**図17**)．

SPARCでは割り込み/例外をトラップと呼び，トラ

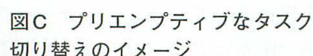
Column 3 割り込みとタスク切り替え

マルチタスクの実現方法として，プリエンプティブ方式というものがある．これはタイマ割り込み(一定間隔で発生する割り込み要求)を契機としてタスクを切り替える方式である．

実行中のタスクは時間が来る(タイマ割り込みが発生する)と割り込みを受け付けて，その実行を強制的に中断し，制御をOSのタスク制御プログラムに移す．OSは中断したタスクのコンテキスト(実行を再現するためのPCやレジスタなどの値)を退避し，次に優先順位の高いタスクのコンテキストを回復して，そのタスクに実行の制御を移す．

このようにして，一つしかないMPUが，複数のプログラム(タスク)を短い時間に少しずつ実行していくことでそれらが同時に動作しているように見せる．これがマルチタスクによる並行処理の正体である．そして，マルチタスク動作を行うためのキーポイントとなるのがタイマ割り込みという割り込みの一種なのである．

三つのタスク(タスクA，タスクB，タスクC)が存在する場合にタスク切り替えが行われるしくみのイメージを**図C**に示す．タイマ割り込み自体は，MPU内部のタイマカウンタにOSが値を設定し，そのカウンタの値が一定値に達すると割り込み要求が発生するというものだ．また，タイマ割り込みが発生するごとにOS内でタイマカウンタは設定し直される．



図C プリエンプティブなタスク切り替えのイメージ

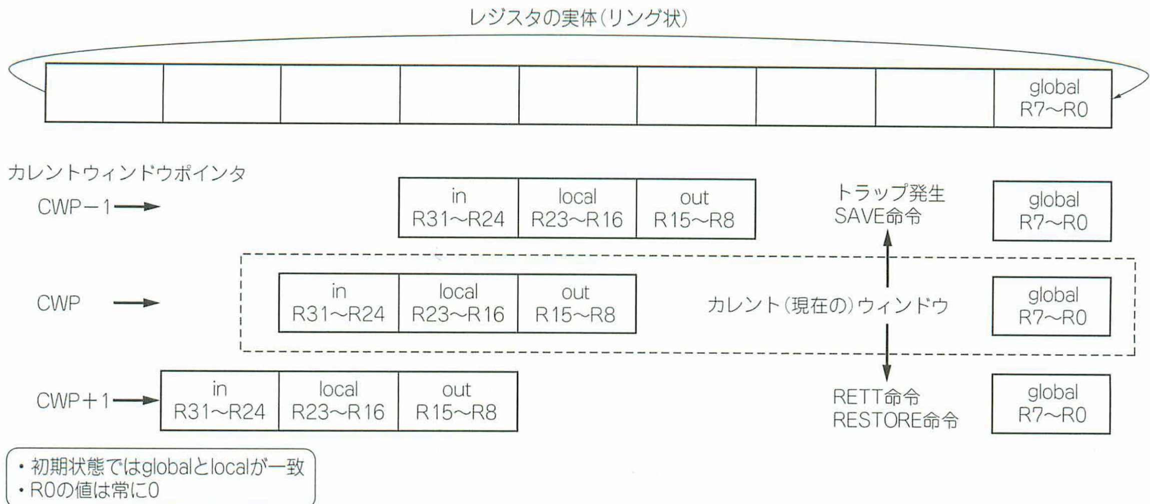


図17　SPARCのウィンドウレジスタの構成

ップも手続き呼び出しと同様の挙動をする．つまり，通常の手続き呼び出しと同様に，トラップが発生するとカレントウィンドウポインタを減少させ，次のレジスタウィンドウを指し示す．そして，トラップを起こした命令のPCとその次の命令のPCを新しいウィンドウの二つのローカルレジスタ(R17とR18)に格納する．

一般的に，トラップハンドラはPSR(Program Status Register)の値を他のローカルレジスタに退避させる．このため，新しいウィンドウのほかの五つのローカルレジスタが使用可能である．トラップは命令によって起因する例外，または特定の例外とは無関係な外部割り込みによって引き起こされる．命令が実行される前，例外や割り込み要求のうちのどれかが発生していると，IUは最高の優先順位をもつものを選択してトラップを発生させる．

▶例外ベクタの構成

トラップが発生すると，各トラップハンドラの最初の4命令が格納された特殊なトラップテーブルをベクタ参照してスーパバイザへ制御を移行する．このテーブルのベースアドレスは，IUのトラップベースレジスタ(TBR)で規定される．

また，トラップテーブル内のオフセットはトラップタイプ(tt)によって決定される．ttはTBRのビット11〜4に格納されるので，トラップ発生時のTBRの値がそのままベクタの値となる．TBRのビット3〜0は0固定なので，トラップテーブルの各エントリでは対応するトラップハンドラに対して16バイトの領域(4命令)が確保されていることになる．

なお，トラップテーブルの半分はハードウェアトラップ用である．残りの半分はソフトウェアトラップ命令(Ticc)によって発生するソフトウェアトラップ用である．ただし，リセットのみは0番地に分岐する．

IUは外部割り込み要求に対しては，割り込み要求レベル(bp_IRL)をPSRのプロセッサ割り込みレベル(PIL)と比較し，bp_IRLのほうがPILより大きいか，bp_IRL=15(NMI)の場合は，プロセッサが割り込み要求をトラップとして受け付ける．

なお，トラップが発生するためには，PSRの割り込み許可(ET)ビットが1であることが必要である．ETビットが0なら，bp_IRL = 15であっても割り込みは発生しない．

▶割り込み/例外の動作

表9に，SPARC(Version 8)のトラップの優先順位とトラップタイプを示す．SPARCでは，トラップが発生すると次のような動作を行う．

1) トラップを不許可(ET←0)
2) 現在のユーザー/スーパーバイザモードが保存される(PS←S)
3) ユーザー/スーパーバイザモードがスーパーバイザに(S←1)
4) レジスタウィンドウが切り替わる(CWP←CWP − 1)
5) トラップを発生したPCを新しいウィンドウのローカルレジスタ1，2に格納する(R17←PC；R18←nPC)
6) 例外や割り込み要求を特定するtt値が書き込まれる
7) リセットなら制御はアドレス0に移行し(PC←0,

表9　SPARCの例外/割り込み要求の優先順位とトラップタイプ

例外割り込み要求	優先順位	tt(Trap Type)	例外割り込み要求	優先順位	tt(Trap Type)
リセット	1	不定	タグオーバフロー	14	0x0A
データストアエラー	2	0x2B	ゼロ除算	15	0x2A
命令参照MMUミス	2	0x3C	トラップ命令	16	0x80～0xFF
命令参照エラー	3	0x21	割り込みレベル15	17	0x1F
Rレジスタ参照エラー	4	0x20	割り込みレベル14	18	0x1E
命令参照例外	5	0x01	割り込みレベル13	19	0x1D
特権命令	6	0x03	割り込みレベル12	20	0x1C
不正命令	7	0x02	割り込みレベル11	21	0x1B
FP不許可	8	0x04	割り込みレベル10	22	0x1A
CP不許可	8	0x24	割り込みレベル9	23	0x19
未実装FLUSH	8	0x25	割り込みレベル8	24	0x18
ウォッチポイント検出	8	0x0B	割り込みレベル7	25	0x17
ウィンドウオーバフロー	9	0x05	割り込みレベル6	26	0x16
ウィンドウアンダフロー	9	0x06	割り込みレベル5	27	0x15
メモリアドレス不整列	10	0x07	割り込みレベル4	28	0x14
FP例外	11	0x08	割り込みレベル3	29	0x13
CP例外	11	0x28	割り込みレベル2	30	0x12
データ参照エラー	12	0x29	割り込みレベル1	31	0x11
データ参照MMUミス	12	0x2C	実装依存例外	実装依存	0x60～0x7F
データ参照例外	13	0x09			

nPC←4)，リセットでないなら制御はトラップテーブルの中に移行する(PC←TBR；nPC←TBR+4)

8) ソフトウェアでトラップ処理を行う
9) RETT(Return from Trap)命令を実行する(CWP←CWP+1；S←PS；ET←1)
10) JMPL命令を実行(実際にはJMPLの遅延スロットにRETTを置く)してハンドラから復帰する．RETTだけでは後続の数命令を実行してしまうため

例1) トラップした命令に復帰

```
JMPL  %R17,%R0
RETT  %R18
```

例2) トラップした次の命令に復帰

```
JMPL  %R18,%R0
RETT  %R18+4
```

● 各MPUの特徴のまとめ

こうして，各アーキテクチャの割り込み/例外処理の実装を見ると，次のようなことがわかる．CISC(というか古いMPU)では，ハンドラのアドレスが格納されたテーブルを参照して分岐先を決定するのに対し，RISCでは割り込み/例外の種類に応じた特定のアドレスに直接分岐するのである(SH-1/SH-2を除く)．これは，少しでもメモリ参照回数を低減して性能向上を図るという，RISCの設計方針が表れたものかもしれない．

まとめ

割り込みや例外というものは，MPUの動き自体は単純なものである．なぜ，そのような機構が提供されているのか，その思想的背景を理解することのほうが難しい．今回の説明で少しはわかっていただけたであろうか(じつは少し不安)．

割り込みといえば，何かの仕事を中断して別の仕事をするというイメージである．この場合，後で中断した処理を再開するために，スタックに戻りアドレスなどの復帰情報を退避させることが前提である．このため，現実の生活では，割り込み仕事が連続して頻発すると，「スタックがオーバフローして，さっきまで何をやっていたかわからないよ」と悲鳴を上げることがしばしばである．これも職業病だろうか．

Appendix 3

クロック周波数の上限は何で決まるのか

高速化技術の基礎

● 高速化とはどういうことか？

「MPUが高速」という場合は，一般には処理性能が高いことを示す．MPUの処理時間は，次の式で表される．

$$処理時間 = \frac{処理に要する総クロック数}{クロック周波数}$$

$$= \frac{処理に要する命令数 \times CPI}{クロック周波数}$$

処理性能が高いということは，この処理時間が短いということである．処理に要する命令数を減らすことで，処理時間を短くするのがCISCのアプローチであった．反面，CPI(Clocks per Instruction)を減らすこととクロック周波数を高くすることがRISCのアプローチであった．最近のMPUはRISC化されているので，RISCのアプローチをメインに考える．

CPIを減らすこと，つまり，IPC(Instructions per Clock)を増やす工夫は，第2章や第3章で説明したパイプライン，スーパースカラなどで行われている．クロック周波数を上げる工夫は，一部はスーパーパイプラインで行われている．しかし，これらマイクロアーキテクチャ的なアプローチはすでに出尽くした感がある．そこでここでは，クロック周波数を上げるという観点で考えてみたい．以下，「高速」という言葉は**クロック周波数が高い**という意味で使用する．

● クロック周波数を規定する要因

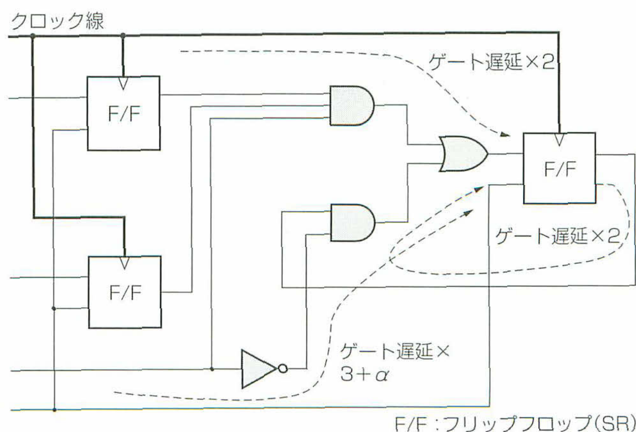
クロック周波数とは，クロックが単位時間に変化する回数を示す．MPUはクロックに同期して動作しているため，クロック周波数が処理速度を規定する．つまり，クロック周波数を高くすればするほど，MPUは高性能になる．しかし，クロック周波数は無条件に高くはできない．それは，MPUの内部回路を電気信号が伝わる時間(電流の速度)に依存するからである．

つまり，クロックが1回変化する時間(これが1周期＝周波数の逆数)に電気信号が移動できる距離が，クロック周波数の物理的な限界である．しかも内部回路には，次のように電気信号の流れを妨げるいろいろな要因が存在する．

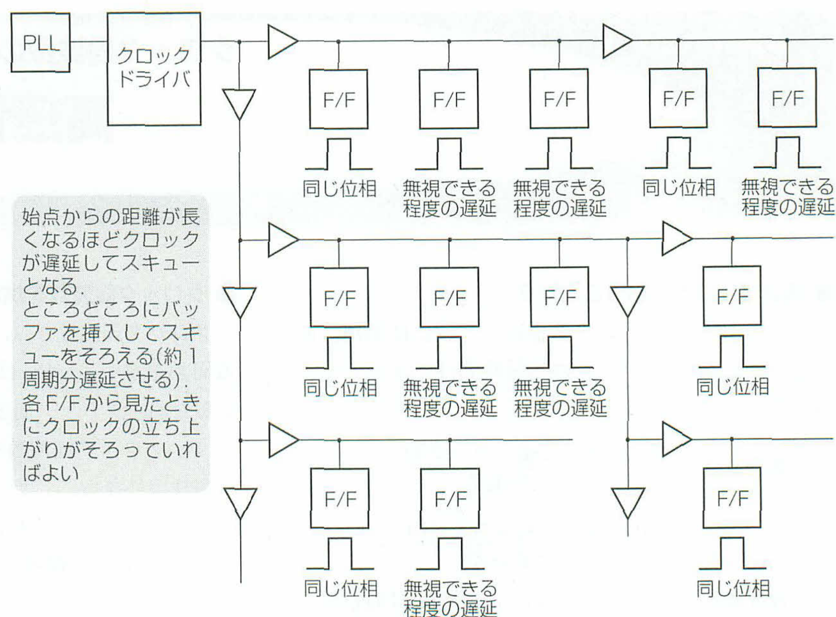
▶ クリティカルパス

クリティカルパス(critical path)とは，二つのフリップフロップ(クロックが供給されるラッチ)間の配線で生じる最大遅延時間のことである．MPUの内部回路はクロックに同期して動くため，クロックの1周期の間にあるフリップフロップから別のフリップフロップに電気信号が伝わらないと誤動作する．

現実には，フリップフロップ間には何段階にもわたってAND/OR/XOR/NOTといった論理ゲートが存在する(**図A**)．電気信号はこれらの論理ゲートを通過するたびに少しずつ遅延(ゲート遅延)が生じる．また，



図A　フリップフロップと論理ゲート



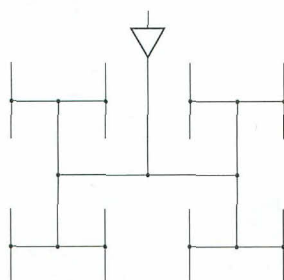
図B　クロックスキュー

配線自身の抵抗によっても遅延(配線遅延)が生じる．これらを合計した遅延時間がクロックの1周期の時間より小さくないと誤動作する．

高速化を実現するための基本は，ゲート遅延を低減するためにフリップフロップ間の論理ゲートの段数を減らすことである．これは論理設計の役割である．あるいは，配線遅延を低減するために配線を短くすることである．これは回路設計の役割である．とはいえ，一つの配線が多くの論理ゲートを通過していると論理ゲート用にある程度の距離が必要なので，配線を短くするためには論理ゲートの段数を減らすことも必要である．

▶ **クロックスキュー**

クリティカルパスはフリップフロップ間の電気信号の遅延によって規定されるが，これは，すべてのフリップフロップに対して，クロックが同じタイミングで変化することを前提としている．しかし，クロックも配線によって伝達されるので，配線の形状や長さによってバラツキを生じる．このクロックのバラツキをなくすために，クロックの配線にバッファを入れたり遅延素子を入れたりして遅延をそろえることが行われる(**図B**)．



図C
H-Tree型のクロック配線

しかし，クロックのバラツキを完全に一致させることは不可能なので，一致させられなかった分が**クロックスキュー**となる．クロックスキューは，フリップフロップを伝わる電気信号から見ると遅延とみなされるので，クロック周波数低下の要因となる．

クロックスキューを揃えるために一般的に使用されるのは，「H-Tree」と呼ばれる手法である．これはクロックラインをアルファベットのH形状に配線することで，クロックドライバからクロックの供給先までの距離をそろえる(**図C**)．IntelのPrescottはH-Tree構造を改良し，Northwoodでは22psであったクロックスキューをPrescottでは7psに低減している．

ARM10の製造で1.2GHzを達成したSumsung社は，H-Tree構造ではなく「メッシュ」構造を採用している．これはクロックをメッシュ状に配線し，その各辺から同時にクロックを供給する手法である(**図D**)．H-Treeに比べるとクロックスキューは小さくなるが，同時に駆動するラインが多いため，消費電力が大きくなる．

▶ **ゲート遅延**

ゲート遅延はトランジスタ遅延ともいわれる．つまり，トランジスタのソースからドレインに電気信号(NMOSトランジスタの場合は電子，PMOSトランジ

スタの場合は正孔)が流れる場合の遅延時間のことである．これはゲートの下に形成されるチャネル長に比例して大きくなる．したがって，ゲート長(やチャネル長)を短くできる微細プロセスを使用すれば，ゲート遅延を低減できる．

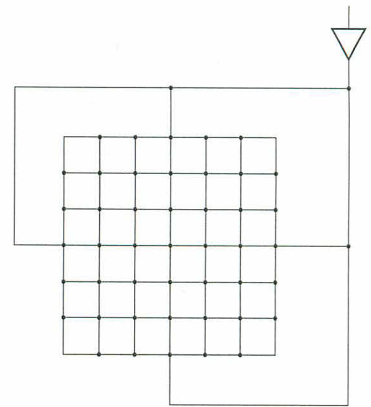
ゲート遅延はトランジスタの製造条件，すなわち，V_{th}(Threshold Voltage，V_tとも呼ぶ)，I_{on}(high drain current)，I_{off}(low off state leakage)などにも依存する．V_{th}とは，ソースからドレインに電流が流れ始める電圧のことで，この電圧が低いほどトランジスタは高速にON/OFF動作する．I_{on}とは，トランジスタがON時の単位長当たりの電流値(通常，μA/μmという単位で示される)のことで，この値が大きいほど遅延が少なくなる．

I_{on}を大きく製造するということは，トランジスタの電流駆動能力を大きく製造することである．I_{off}とは，トランジスタがOFF時のソースからドレインに流れるリーク電流のことで，消費電力を下げるためにはI_{off}の低減が必須である(I_{off}の単位は，通常nA/μmなので，I_{on}に比べると非常に小さいのだが)．

リーク電流はトランジスタのON電圧を邪魔するので，I_{off}の低減はV_{th}の低下を可能にする．リーク電流を抑えるためにSOI(Silicon On Insulator)という技術などが提案されている．

以上のことから，微細プロセスでゲート長を短くし，V_{th}が低く，I_{on}が大きいトランジスタを製造すれば，高速なON/OFF動作が可能になる．近年，THzトランジスタ(TeraHertz Transistor)，すなわち1THz以上のクロック周波数で動作するトランジスタの発表が花盛りであるが，これはゲート遅延が1ps以下であることを示しているにすぎない．つまり，ゲート遅延の逆数をクロック周波数とみなしているのである．本来のクロック周波数は回路の配線遅延などにも依存するので，それがそのままMPUの動作周波数にはならない．

ところで，温度が上昇するとチャネルの熱抵抗が増加するのでゲート遅延が増加する．そこで，トランジスタを冷却することでクロック周波数を高めることも考えられる．半導体の製造メーカーは，ある決められた温度範囲内で動作クロック周波数を保証しようとするため，無理やり冷却することでクロック周波数を向上させることは想定していない．しかし，PCの自作などのクロックアップでは，メーカーの保証外ではあるが，とにかく冷やすというのが常套手段である．



図D メッシュ型のクロック配線

▶ 歪みシリコン(Strained Silicon)

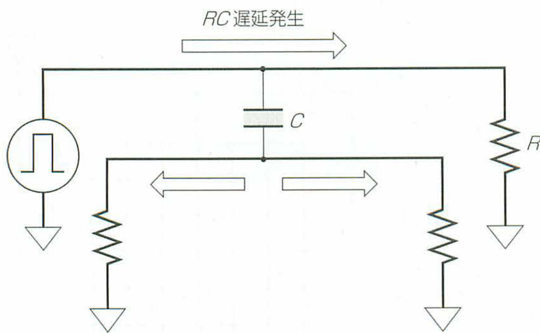
MOSトランジスタの微細化にともない，ゲート酸化膜が薄くなるとリーク電流が増加するため，それを抑えるためのhigh-k(誘電率の高い)絶縁膜の研究がさかんである．しかし，high-k絶縁膜では，SiO_2に比べ，電気信号(電子や正孔)の移動速度が低下してしまう．そこで，電気信号の移動度を高める手段として**歪みシリコン**が注目されている．

シリコン単体より格子定数の大きいSiGe結晶の上に薄いSi膜を作ると格子が引き延ばされて歪むのだが，この状態では通常のシリコンに比べて電子，正孔の移動度が向上することが知られている．しかし，無理な力を加えて製造するので格子欠陥ができやすいという問題がある．

また，歪みシリコンによるキャリア移動度の向上度合いが電子と正孔で差があることや，NMOSトランジスタではしきい値が低下するため，回路設計が難しいという問題もある．そのためか，歪みシリコンを利用したMOSデバイスが発表されたのは2001年と意外に新しい．本格的な実用化はまだ先の話である．

……と思っていたところ，Intelはゲート長50nm(ゲート酸化膜厚1.2nm，設計ルールは90nm)のCMOSプロセス「P1262」で歪みシリコンを採用することを明らかにした(2002年8月13日)．Intelの試算ではトランジスタの電流速度が10～20%向上する．このP1262は，Pentium4系のPrescottなどに適用されている．

歪みシリコンといえば，IBMがいちばん先行していると考えられている．そのIBMですら65nm世代から適用と表明していたが，後発のIntelが，1世代早い90nmから適用すると発表して注目を集めている．なお，IBMによると歪みシリコンを用いた場合の動作



図E　配線の寄生容量

速度は，最大35％向上するという．

▶ 配線遅延

遅延はトランジスタだけでなく，配線自身によっても発生する．配線を伝わる電気信号の遅延は，配線抵抗Rと配線容量Cの積で決定される（RC遅延）．電気信号はデバイス中を伝達する際に，配線にぶら下がっている容量（キャパシタンス＝コンデンサ）を充放電させながら伝搬される（図E）．

配線抵抗を下げるためには配線幅を広くし，配線の厚さを厚くしなければならない．しかし，それでは微細化に反する．通常のMPUではアルミニウムで配線を行うが，最近は，より電気抵抗の小さい銅を用いて配線するのが流行である．銅を使用すれば，細い配線にしても電気信号がスムーズに流れる．しかし，銅配線を使用するためにはダマシン（damascene）プロセスという新しい配線形成方式が必要になるので製造の手間がかかる．このため，現在では真に高速を要する場面でしか使用されない．

また，製造プロセスの微細化により，配線の寄生容量が増加する傾向にある．配線間の寄生容量を減らすためには，配線間隔を広くする必要がある．これらはプロセスの微細化とは逆行する．その解決策が多層配線である．これは，高集積化された配線層を，層間膜を挟んで何層にも積み重ねて配線系を実現するものである．

現在，層間膜としては二酸化シリコン（SiO_2）が主流である．しかし，寄生容量を低減するために，さらに誘電率の低い（low-k）材質であるHSQ膜，アモルファスカーボン膜などが注目されている．SiO_2の誘電率は4.1であり，一般にlow-k材質といえば誘電率が3.0以下の材質を意味する．

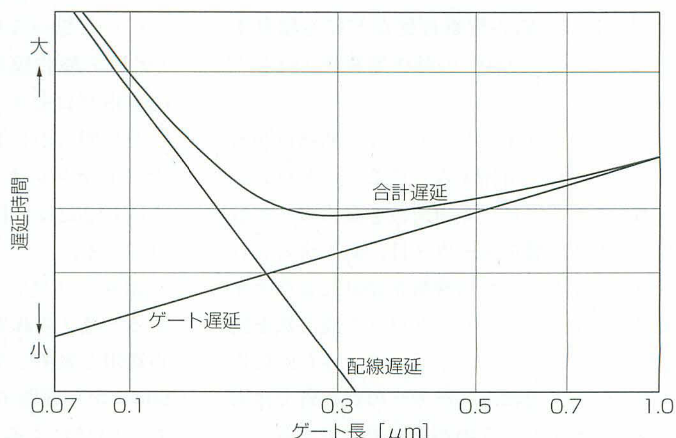
配線材料の低抵抗化は，銅配線の採用でほぼ物理的限界に来ている．このため，層間絶縁膜材料の低誘電率化は，高速配線実現の要としてますます重要になっている．

従来，回路の遅延はゲート遅延が大勢を占めていたが，製造プロセスが微細化するのにともない，配線遅延を無視できなくなってきた（図F）．製造プロセスの微細化は，ゲート遅延を低下させるが，反対に配線遅延を増加させる．その境目は，ゲート長が0.18 μm辺りにあるといわれている．

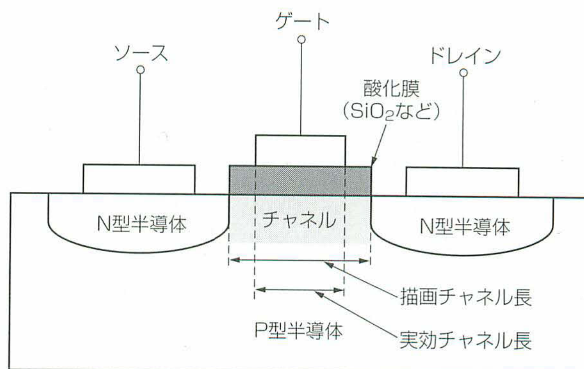
▶ 動作電圧

最近，製造プロセスの微細化にともない，**動作電圧**も低下する傾向にある．基本的に，CMOS回路は面積が小さいほど低い電力で動作できる．また，CMOS回路は電圧が高いほどスイッチング速度が速い．あるいは，CMOS回路の動作速度は，NMOS，PMOSトランジスタの電流駆動能力に依存する．

電流駆動能力は，大雑把にいうと，ゲートにかかる電圧とV_{th}の差に比例する．このため，動作電圧を高くすれば高速動作が可能である．しかし，あまり大き



図F　ゲート遅延と配線遅延



図G　NMOSトランジスタの断面

な電圧を加えるとトランジスタが破壊されてしまう．製造プロセスが微細化するにしたがって，この限界電圧(絶対最大定格という)も低下する傾向にある．また，微細化により寄生容量の影響が支配的になり，電圧の増加に対する高速動作への寄与も飽和状態にある．したがって，むやみに高い電圧を加えればよいというものでもない．

ところで，動作電圧を上げるということは消費電力が増大することでもある．むしろ，動作電圧の決定には，動作速度よりも消費電力の限界のほうが支配的になっている．したがって，近年の研究では，動作速度をある程度維持しつつ，いかに消費電力を下げるかが課題になっている．つまり，基本的には動作電圧を下げるとともに，V_{th}も下げるという手法が採られる．V_{th}を下げると，リーク電流が増大する傾向にあるので，何らかの対策が必要なのは上述したとおりである．

▶ ***IR*ドロップ**

製造プロセスの微細化が配線の抵抗を増大させることはすでに述べた．このため，電源からトランジスタまでの距離が長くなると，配線抵抗によって電圧降下(*IR*ドロップ)が発生する．これは，トランジスタの動作電圧を下げることに等しく，動作速度の低下につながる．このため，半導体回路では電源構造の配置も重要になる．

● 製造プロセスの微細化

▶ **プロセスルール**

半導体の製造プロセスは「0.13 μm」などと長さを指定して表現する．この値が小さいほど微細な製造プロセスだということができる．長さの単位は，2001年くらいまではμm(マイクロメートル＝ミクロン)が主流であったが，2002年以降はプロセスの微細化が進んだためnm(ナノメートル)で表現することが多くなった．

この長さは，トランジスタのゲートの幅を表すものである．ソースとドレイン間のチャネルの長さにも等しいので，**チャネル長**ともいう．そして，このチャネル長には，リソグラフィ技術(写真のようにマスクパターンをシリコンウェハに露光する技術)で物理的に形成される**描画チャネル長**と，ソースとドレインに実際に電圧を加えた場合に電気的に形成される**実効チャネル長**の2種類がある(**図G**)．現実には，描画チャネル長のほうが実効チャネル長よりも大きい．

製造プロセスを表す場合に，どちらの基準を使用するかは国やメーカーで異なる．一般的には，米国メーカーは描画チャネル長を使用し，日本メーカーは実効チャネル長を使用する傾向がある．しかし，このような状況は混乱を招くので，世界的には描画チャネル長で基準を統一する動きがある．将来的には，すべて描画チャネル長に統一されると予想される．具体的には，描画チャネル長が製造プロセスの世代を表し，実効チャネル長でゲート長を表すことになる．

ゲート長が短いほど，高い動作周波数を実現できる．キャリアが移動する距離が短いほど，ゲート遅延が少なくなるからである．

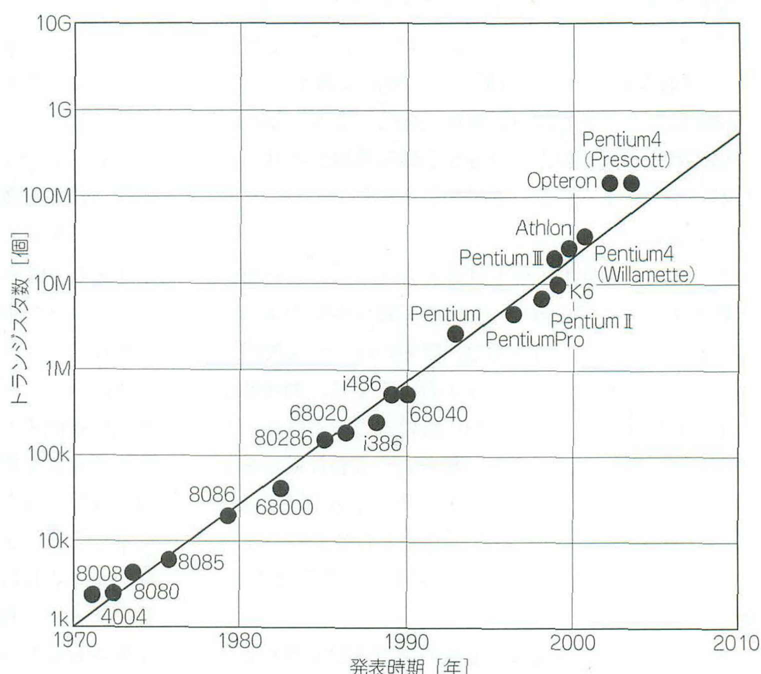
表Aに，Intelの製造プロセスの変遷を示す．動作周波数はPentiumを基本としているが，Pentium4になってパイプラインのステージ数が増えたので，2000年以降は上限の動作周波数が一気に上昇している．

▶ **ムーアの法則**

製造プロセスが微細化するにともなってトランジスタの面積は小さくなる．このため，MPUの1チップに集積できるトランジスタ数は相対的に多くなる．トランジスタ数が多いということは，実現できる機能が多いということであり，同時に微細化により動作周波数も上昇するので，MPUを使えば何でもできるという世界が近づきつつある．このような半導体の進歩は，

表A　Intelの製造プロセスの変遷

導入年	製造プロセス（世代）	ゲート長	動作周波数（Hz）	配線	チャネル	ゲート絶縁膜	ゲート電極	動作電圧
1993	0.50 μm	0.50 μm	150～200M	Al	Si	SiO2	多結晶Si	3.3V
1995	0.35 μm	0.35 μm	233～300M	Al	Si	SiO2	多結晶Si	2.5V
1997	0.25 μm	0.20 μm	300～533M	Al	Si	SiO2	多結晶Si	1.8V
1999	0.18 μm	0.13 μm	500～2000M	Al	Si	SiO2	多結晶Si	1.5V
2001	0.13 μm	70nm	1.3～2.4G	Cu	Si	SiO2	多結晶Si	1.3V
2003	90nm	50nm	3.0～5.0G?	Cu	歪みSi	SiO2	多結晶Si	1.1V
2005	65nm	30nm	10G?	Cu	歪みSi	SiO2	多結晶Si	0.85V
2007	45nm	20nm	20G?	Cu	歪みSi	High-k	金属	0.70V
2009	32nm	15nm	50G?	CU	歪みSi	High-k	金属	0.60V
2011	22nm	10nm?	100G?	?	歪みSi	High-k	金属	?



図H　x86系MPUのトランジスタ数
（2007年には，動作周波数20GHz，トランジスタ数10億個に到達予定）

Intelの創立者の一人であるGordon Moore氏が1975年に提唱した「チップに集積可能なトランジスタ数は18～24か月ごとに倍増する」というムーアの法則にも示されている（**図H**）．

しかし近年，このムーアの法則の有効性がいつまで維持できるかが注目を集めている．「技術的にあと2～3世代（6年）が限界」，「18～24か月というペースに限界が出てきた」，「限界はまだまだ，問題は手段である」など，さまざまな見方がある．ところが，2001年になってから，この限界説が和らいできている．少なくとも後10年，つまり今後5世代に関してはムーアの法則は維持されるだろうと見るアナリストが多くなった．

ムーアの法則自体は経験則なので，根拠があるわけではない．半導体がムーアの法則にしたがって進化しているというよりも，半導体メーカーがムーアの法則を維持するために，それを目標として製品開発をしているといったほうが正確であろう．

▶どこまで微細なパターンを描画可能か

現在の半導体は，遠紫外線を使用するリソグラフィ（lithography）技術により，透明な石英板の上にクロム（Cr）でトランジスタや配線のパターンが形成されているマスク（レチクル：reticle）の模様を，シリコンのウェハに転写して回路を生成する．この技法では，今後の1～2世代に相当する100nmまでのパターンしか対応できないとされている．2002年では，半導体

メーカーは0.13 μmプロセスが主流であり，限界に近づきつつある．

リソグラフィの光源として，0.13 μmまではKrFエキシマレーザが使用され，0.13 μm～0.10 μm(＝100nm)まではArFエキシマレーザが有力候補とされている．これに代わる新しいリソグラフィ技術がなければ，半導体メーカーは2004年か2005年に壁に突き当たり，それ以上MPUの集積化(＝高速化)ができなくなってしまう．その候補としては，F_2エキシマレーザ(波長：157nm)，等倍X線，縮小X線，電子ビームなどがある．それぞれ一長一短があり決定版はない．

▶EUV(超紫外線)リソグラフィ技術

新しいリソグラフィ技術として期待されているのがEUV(Extreme Ultra Violet)リソグラフィ技術である．KrFリソグラフィで使用する波長は246nm，ArFリソグラフィで使用する波長は193nmであるが，EUVはこれよりもずっと短い波長を使用する．EUVでは最高で30nmという微細な露光が可能になり，2005年～2006年には10GHzのMPUを実現することが可能になるといわれている．

Intel，Motorola，AMD，Micron Technology，Infineon Technologies，IBMなどが参加している民間の業界コンソーシアムEUV LLC(Extreme Ultra Violet Limited Liability Company)は，EUVリソグラフィ技術を用いたMPUを搭載するプロトタイプ機が完成したことを2001年3月に発表した．これら各社には，メンバ以外の半導体メーカーに先駆けてEUV装置を導入する権利が与えられている．

2002年4月，Intelがオランダの半導体製造装置メーカーにEUVリソグラフィ用の装置のβ版を発注したことが明らかになった．納品は2005年下半期とかなり先のことだが，新しいツールの導入には各種の微調整に時間がかかるため長いリードタイムが必要になるということである．EUVを採用したMPUの出荷は5年以内に始まる見通しという．

▶EB(電子ビーム)リソグラフィ技術

IBMは，EUVリソグラフィの重要性を認めてはいるが，EB(Electron Beam)リソグラフィ技術もニコンとの共同事業としてサポートし続ける予定だという．両社は，「Prevail」と名付けたEBリソグラフィ技術を共同開発している．2003年時点でPrevailは開発のα段階にある．

EBリソグラフィ技術は，EUVリソグラフィより波長が数桁短いので，EUVよりもはるかに細かいパターンを描くことが可能だが，一度に描画できる面積がEUVよりも小さい．したがって，EBを使った大量生産はEUVと比べて非常に遅い．

業界では，この二つの技術は競合すると見られがちであるが，IBMは二つの技術を補完的に組み合わせて応用することを考えているらしい．EUVでMPUの大きな構造を一度に描き上げ，電子ビームでより微細かつ重要な部分を描くというのがその計画だという．

● フィンFETと多重ゲートトランジスタ

FETとは電界効果トランジスタ(Field Effect Transistor)のことである．ソースとドレイン間の電流をゲート電圧で制御する通常のトランジスタである．従来のFETはプレーナ型と呼ばれ，シリコンに不純物を注入してソースとドレインを形成し，ソースとドレイン間のチャネルの上部に絶縁膜を挟んでゲート電極を配置する．つまり，平面状に，ソース-チャネル-ドレインが並んでいる〔図1(a)〕．

FinFETは，プレーナ型FETを拡張し，立体構造にすることで，高速動作性能と省電力性能を高める．絶縁基板上にソース-ドレインが直方体状に形成され，その間にあるチャネル部の両側を絶縁膜を挟んでゲート電極が包み込む〔図1(b)〕．ゲート構造がフィン(ひれ)のような形状をしているところからFinFETと呼ばれる．このFinFETは，高速化，高集積化，低消費電力化を実現できる次世代のトランジスタ構造として注目を集めている．

FinFETは，1999年11月22日カリフォルニア大学のバークレー校でコンピュータサイエンスを担当するチェンミン・フー(C. Hu)教授らのグループが発明したとされている．その設計技術を用いれば，1チップ上に従来の400倍のトランジスタを搭載できるという．

フー教授らが開発したのは，ゲートをチャネルの両側にまたがるフォーク形にして，チャネル電流を効果的にコントロールする方法である．フー教授の例えを借りれば，「出血を止める場合，両側から血管を挟めば，片方向から押すよりはるかに効果がある」ということである．研究グループが1999年7月に製造したプロトタイプのゲート長は18nmである．

FinFETと同等の発想をもつトランジスタの発表としては，2001年12月にIBMがダブルゲートトランジスタの実用化に向けた開発成功を発表，2002年6月にはTSMCが35nmのゲート長で動作可能な高性能FinFETの開発について発表した．さらに，2002年9月には，AMDがゲート長10nmのFinFETを試作した

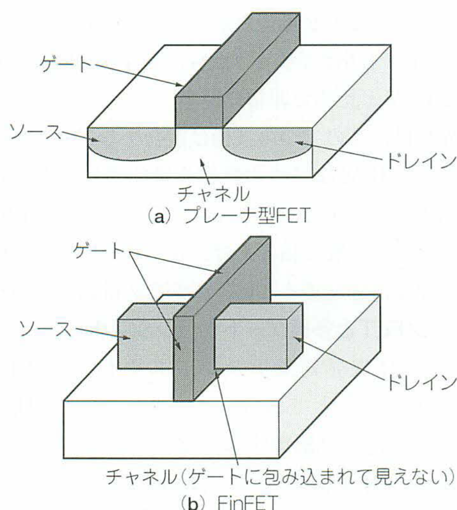


図1 プレーナ型FETとFinFET

ことを発表，続いてインテルがトライゲートトランジスタの開発を発表した．

FinFETは，現在の半導体製造技術で容易に製造できることに加え，プレーナ型FETと比較して次のような利点がある．

1) ゲート面積を大きくできるため，チャネルを流れる電流量を大きくできる．すなわち，高速なスイッチングが可能になる．同じトランジスタ能力なら，チャネル電流量を少なくすることができるので，低消費電力を実現できる．

2) プレーナ型では微細プロセスになるほど短チャネル効果の抑制が困難になり，リーク電流が増加する．FinFETではチャネルをゲートで包み込むため，ゲート絶縁膜をそれほど薄くしなくてもチャネル部にキャリアを集めやすく，高速動作，あるいはゲートリーク電流の制御が容易になる．

なお，短チャネル効果とは，ゲート長(チャネル長)が短い場合，ソースとドレインが近づき過ぎるとゲートを閉じていてもソースとドレインの間でリーク電流が流れてしまう現象を指す．

3) プレーナ型はチャネル部がチップの表面に平行であるため，チップの単位面積当たりの駆動電流量は一定である．しかし，FinFETではチャネル部がチップに対して垂直であり，フィンの高さを上げることでチップの単位面積当たりの駆動電流量を容易に増やすことができる．これも高速なスイッチングに寄与する．

4) 3次元的にトランジスタを形成するため，トランジスタ面積を小さくでき，単位面積当たりのトランジスタ数を増加させることができる．

2003年6月12日，Intelは30nmプロセスのトライゲートトランジスタの開発に成功したと発表した．従来のゲート長は60nmだったが，今回30nmで製造(プロセスルールは90nm)できたことで，トライゲートトランジスタは研究段階を終えて開発段階に到達したと発表した．1.3V動作時のNMOSでオン電流1.23 mA/μm，オフ電流40nA/μmを達成しており，この電流駆動能力は非プレーナ型では世界記録としている．

Intelは，2007年に45nmプロセスでトライゲートトランジスタの実用化を目指すという．また，他社のダブルゲートトランジスタやFinFETと比べ，ソースとドレインを形成するシリコン層の厚さはプレーナ型ではゲート長の1/3倍が限界，ダブルゲートではゲート長の2/3倍が限界とし，トライゲートトランジスタではゲート長と同じ長さにできる(3/3倍)ので製造が容易で大量生産に向くとしている．

2003年9月18日，AMDは，東京で開催された国際固体素子・材料カンファレンス(SSDM)で実験的なトライゲートトランジスタを紹介した．このトランジスタには，完全空乏型SOI技術，金属(ニッケルケイ化物)ゲートが採用されている．研究をしているという以上の意義はないが，このトランジスタは2007年までに実用化されるという．

また，2003年11月10日，Motorolaは二つのゲートがそれぞれ独立に機能するデュアルゲートトランジスタ「Multiple Independent Gate Field Effect Transistor」(MIGFET)の開発に成功したと発表した．通常のマルチゲートトランジスタは，複数のゲートが同じように動作する．しかし，MotorolaのMIGFETでは，ゲートが電気的に分離された状態にあるため，開発者の意図に応じて同時動作や個別動作が可能なので，性能向上と電力低減に多大な利用法を提供する．

たとえば，トランジスタ数百万個が必要な複雑な計算も，約半数のトランジスタで実行できるようになるという．Motorolaは，この技術の製品適用時期を明言していないが，別のメーカーは，このような技術は2007年から2008年の45nmプロセスで実現されるだろうとしている．

● 最近の話題から

▶ 配線技術の新機軸

ZDNetによると，2002年6月19日，プリンストン大学のStephen Y. Chou氏らは，シリコンチップに10nm規模の配線パターンを直接プリントする手法を開発したと発表した．

「Laser-Assisted Direct Imprint」(LADI)と呼ばれるこの手法は，酸化シリコン(SiO_2)でできた金型をシリコン基板に押しつけ，その上から波長308nmのエキシマレーザを200億分の1秒間パルス照射することで，金型の周りのシリコンを加熱融解させて再度凝固させる．凝固までの時間は250ns．線幅140nmの加工に成功しているという．

この技術により，製造ステップを減らすだけでなく，ナノ秒単位でプリントプロセスを終えることができる．そのため，従来のエッチング作業は不要になる．また，シリコンチップのトランジスタ集積度は従来の100倍となり，ムーアの法則をあと20年延命させることができるという．

従来のリソグラフィ技術では，露光という非接触な方式を使っているため，フォトマスク(レチクル)は数千回以上の使用に耐える．しかし，今回の技術は物理的な接触が行われるため，金型の耐久性が気になるところである．

また，金型の加工時間にかなりの時間を要すると想像されるのと，具体的な金型の製造方法がはっきりしないので，一般な技術になるか否かは不明である．

▶65nmプロセスのSRAM

2003年11月24日，Intelは65nmプロセスのSRAMの製造に成功したと発表した．65nmプロセスにするだけで，性能は40～50％高速化するという．65nmプロセスでは，消費電力を抑えるために，歪みシリコンとlow-k絶縁膜を使用しているという．

なお，65nmプロセスではトライゲートトランジスタや金属ゲート，high-k絶縁膜は採用しない．こうした技術は，2007年に登場する45nmチップに採用される可能性があるということである．これは，当初の予定通りである．

また，Intelの65nmプロセスでは，IBMなどが研究しているSOI技術も採用されないという．インテルによれば，SOIでは取り立てて性能向上は見られなかったそうである．

▶ムーアの法則の終焉

2003年11月，Intelの技術者によりProceedings of the IEEE(Institute of Electrical and Electronics Engineers)に「Limits to Binary Logic Switch Scaling－A Gedanken Model」と題した論文が掲載された．これは2018年に登場するであろう16nmプロセスでムーアの法則が終わりを告げるというものである．

トランジスタが微細化し，ゲート長が5nm(これは16nmプロセスに相当)程度になると，ゲートに電圧をかけなくてもトンネル効果でソースとドレイン間に電流が流れるようになる．これではスイッチとしてのトランジスタの効果がなくなる．

また，物質から電子を取り出せる限界は1.5nmということから，ゲート長の限界は1.5nmといわれている．このような限界のためトランジスタの微細化には終点があり，それを避けるために，トランジスタは3次元的な階層構造を採って集積度数を上げようとするだろうといわれている．

▶浸漬露光システム(Immersion Lithography)

90nmプロセスにおける露光技術の光源は，ArFエキシマレーザ(波長：193nm)であり，この解像度は約97nmである．この光源は，技術の工夫で65nmまでは耐えられるが，その次の45nmプロセスでは対応できない．そこでF_2レーザ(波長：157nm)が考えられているが，なかなか実用化にならない．その対抗馬として有力視されているのが浸漬露光システムである．

浸漬露光システムとは，光源のレンズとウェハの間に蒸留水を満たすことで光の屈折率を高め，レンズの開口数を改善して解像度を上げる技術である．これによりF_2レーザの波長は131nm相当になり，解像度は約53nmになる．これは45nmプロセスに対応可能な解像度である．32nmプロセスまで可能ではないかという研究者もいる．しかし，原理は簡単であるが，ウェハをどう動かすかという問題やフォトレジストの問題があり実用化は難しいと思われていた．

ところが，2003年になって実用化の兆しが出てきた．この分野はASML，キヤノン，ニコンが共同で開発を行っている．2003年12月に開催されたSemicon Japanで，AMSLは半導体メーカーの実験用に浸漬露光装置を販売するとしている．また，ニコンも2006年には量産用の浸漬露光装置を販売すると発表して注目を浴びている．

なお，Intelは45nmプロセスまではF_2レーザを使用し，その先はEUVやEBといった光源を使用するとしている．しかし，32nmプロセスでの浸漬露光採用の可能性も匂わせている．

＊　　　＊

ここでは，主としてトランジスタレベルでの高速化技術をまとめてみた．この分野は日進月歩なので，まだまだ技術革新を遂げていくと考えられる．

第7章
マイクロプログラミング方式のCISCからVLIWの動作まで

マイクロプログラミングとVLIW

ここではマイクロプログラミングとVLIWについて取り上げる．マイクロプログラミングは最近では見られなくなったMPUの実装方式だが，あえて取り上げるのには理由がある．それはVLIWとの関係である．どのような関係があるのか，それは本章を読んでもらえばわかると思う．いまどき時代錯誤では，といわずに読んでみてほしい．

1 マイクロプログラミングの概念

● マイクロプログラミングとは

マイクロプログラミングの概念は1951年にケンブリッジ大学のM.V.Wilkesによって提案された．その目的はコンピュータの制御系を効率的に設計することである．

図1にコンピュータ(MPU)の基本的な構成図を示す．プログラム(命令)は主記憶(キャッシュの場合もある)からバス制御ユニットによって読み出され，命令デコーダでデコードされる．そして，そのデコード情報をもとに命令が実行(処理)される．この実行制御部の設計を効率的に行うのがマイクロプログラミングである．

MPUの実行はマイクロ操作(micro-operation)と呼ばれる基本操作の組み合わせによって実現されている．制御回路はマイクロ操作を制御する信号を次々と発生する．**図2**にそのようなMPUの実行制御部の概念図を示す．デコードした結果をもとに，(1)レジスタ番号を指定する，(2)演算の種類を指定する，(3)レジスタを選択する，(4)レジスタの内容を演算器に与える，(5)演算結果をレジスタにライトバックするといった操作を実行することで命令が実行できるが，これらの操作がマイクロ命令である．初期のMPUや最近のRISCでは，論理回路の組み合わせでこの制御信号を生成している．これがワイヤードロジック(wired logic)と呼ばれる方式である．ワイヤードロジックの設計は非常に複雑であり，設計ミスをした場合の修正も容易ではない．

マイクロ操作を実現するためのマイクロ命令というものを考える．つまり，マイクロ命令は同時に発生する制御信号に関する情報をまとめて，命令の形式に収める．こうすることで，MPUの実行はマイクロ命令

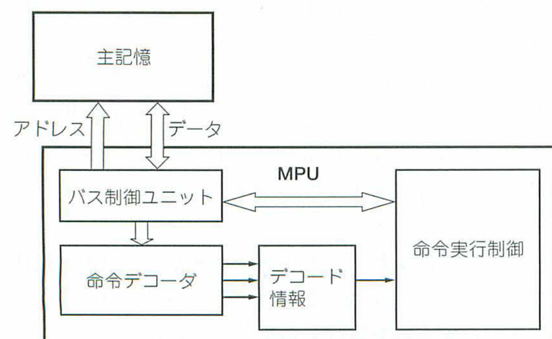


図1　コンピュータ(MPU)の基本的な構成図

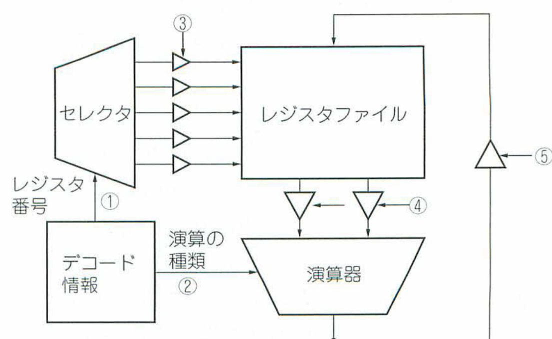


図2　MPUの実行制御部の概念図

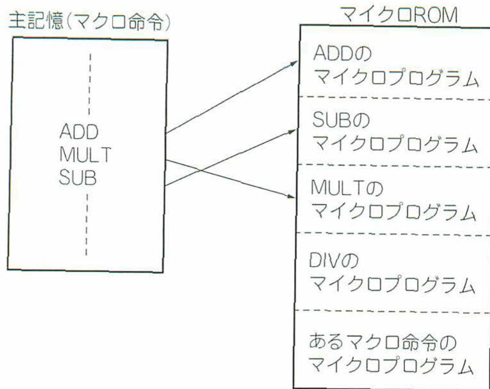


図3 マクロ命令とマイクロプログラム

の系列として定義できる．一般に，MPUのプログラムは命令が並んだものだが，その個々の命令の処理は対応するマイクロ命令を実行する(複数のマイクロ操作を同時に行う)ことで実現できる．これをマイクロプログラムと呼ぶ．マイクロプログラムはマイクロコードと呼ばれることもある．意味的には，ハードウェアとソフトウェアの中間にあることからファーム(やや硬い)ウェア(firmware)と呼ばれることもある．また，マイクロプログラムでMPUの実行を制御する方式をマイクロプログラム制御方式，あるいはマイクロプログラミング方式と呼ぶ．なお，マイクロ命令に対応して，通常の命令はマクロ命令と呼ばれる．

マイクロプログラム制御方式において，マイクロプログラムは適当な記憶装置(通常はROMなのでこれをマイクロROMと呼ぶ)に格納される．そして，マクロ命令はマイクロROM内のアドレスがエンコードして格納されているとみなすことができる．つまり，あるマクロ命令がデコードされるとマイクロROMのアドレスが決定される．同時に，オペランドの種類や演算の種類がデコードされて保持される．命令の実行ステージでは，そのマイクロROMからマイクロ命令が順次読み出されて実行される(**図3**)．

図4にマイクロプログラム制御でマクロ命令が実行されるようすを概念図で示す．イメージとしては，メモリ(マイクロROM)，デコーダ，実行部(マイクロ操作生成)があることから，MPUの内部にさらに小型のMPUがある感じである．そもそも，歴史的に見れば，最初のMPUであるIntelの4004はプログラムを格納するROMの内容を交換することで，種々の電卓に対応しようとした．マイクロプログラムもマイクロROMの内容を交換することで種々の命令セットに対応できる．この奇妙な一致は偶然ではあるまい．

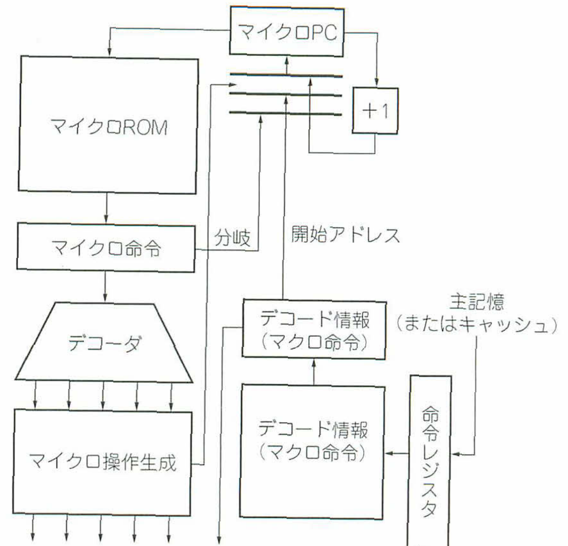


図4 マイクロプログラム制御の概念図

マイクロプログラム制御方式の利点は，制御部の設計がすっきりしたものになることである．MPUの制御論理はマイクロプログラムで実現されるので，その論理設計はマイクロプログラムを作成することが中心となる．マイクロプログラムが完成する前に制御部のハードウェアを並行して(早期に)設計できる．これは命令セットアーキテクチャが定まっていない状態でもMPUの設計を始めることができることを意味する．あるいは，設計の後半で仕様変更が生じた場合でも，マイクロプログラムを変更すれば，ハードウェアの変更なしで対応できる．その意味で命令の追加も容易である．

さらに，マイクロROMの内容を変更することでMPUの制御論理を容易に変更できる．極端な話，一つのハードウェアで複数の命令セットを実行すること(別のMPUのエミュレーション)が可能になる．マイクロプログラム次第でどのようなアーキテクチャのMPUにもなることもできるのだ．

● マイクロ命令の詳細

以下では，マイクロROMに格納されるマイクロ命令について説明していく．

マイクロ命令は，通常，次のような情報から構成される．

(a) 1ステップの間に同時に実行されるマイクロ操作を指定する情報
(b) 次に実行するマイクロ命令を決定する順序制御に関する情報
(c) マイクロ命令で使用する定数

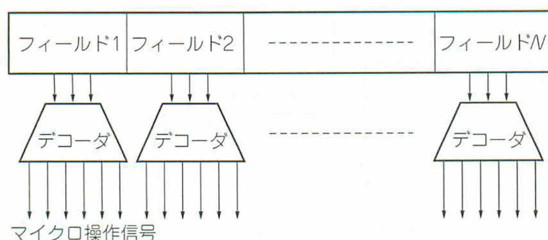


図5　水平型マイクロ命令

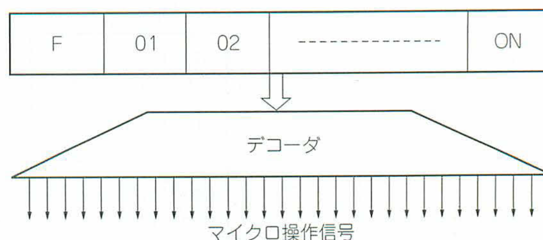


図6　垂直型マイクロ命令

また，マイクロ命令はこれらの情報の表現形式によっていくつかの方式に分類することができる．大別すると，水平型(horizontal type，function/field type)と垂直型(vertical type，machine code type)の二つである．

(1) 水平型マイクロ命令

水平型のマイクロ命令は，マイクロ命令の各ビットが処理装置のゲートなどの制御点に1対1で対応している点に特徴がある．しかし当然ながら，処理装置が複雑になると，制御点も多くなり，マイクロ命令に必要なビット数も増加し，実用的でない．通常はエンコードすることでビット数を減らす方法を採っている．具体的には，マイクロ操作信号を適当なグループに分けて，各グループごとにエンコードしてビット数を削減する．この場合，マイクロ命令を実行するためには各グループ(フィールドという)ごとにデコーダが必要になる(**図5**)．マイクロプログラムの作成者は，各サイクルごとのハードウェアの動き(各制御点の状態)を考慮してコーディングしなければならないので，かなりの技術を要求されることとなる．

(2) 垂直型マイクロ命令

垂直型のマイクロ命令は，どちらかといえばマクロ命令に近い．つまりマイクロ操作のエンコードを複雑にすることにより，マイクロ操作に1対1に対応するという感覚がなくなってしまっているのである．**図6**に，その1例を示す．ここでFは操作コードであり，実行する動作を示す．O1，O2，…，は操作オペランドフィールドで，演算回路の入力や結果の行き先を示す．マイクロプログラムの作成者にとっては，マクロ命令のプログラミングを行う感覚でコーディングできるので親しみやすい．また，アルゴリズム記述向きの形式なので保守もしやすい．

垂直型のマイクロ命令は，ハードウェアとしては同時動作が可能であっても，エンコードの都合により，一つのマイクロ命令では同時に指定できないことがある．したがって，垂直型では水平型に比べてマイクロプログラムのステップ数が増加する傾向にある．高度なエンコードによってマイクロ命令のビット数を削減した反面，マイクロプログラムの容量増加につながっている．

もっとも，現実のマイクロ命令は，水平型，垂直型の区別ができないものが多いのも事実である．両者の利点を折半した中間形式のマイクロ命令も多いらしい．この中間形式のものを対角型(diagonal type)ということがある．

● マイクロプログラムが実行されるようす

図7にマクロ命令実行時のマイクロ命令が実行されるようすを示す．マクロ命令のデコードが終了するとマイクロROMへのアクセスが開始され，次のクロックでマイクロ命令が出力される．**図7**ではマイクロ命令の読み出しと同時にデコードが行われ，すぐにマイクロ操作が実行される場合を示している．また，マイクロROMへはパイプライン式に毎クロックごとにアクセスするものとし，マイクロ命令も毎クロックごとに読み出せるものとしている．**図7**ではマクロ命令が4ステップのマイクロ命令で実現できる場合を示しているが，最初のROMの読み出しに1ステップかかるので，マクロ命令の実行には，実質5クロックかかる．

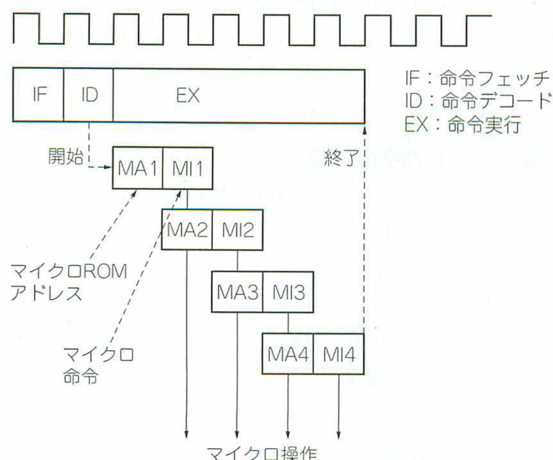


図7　マイクロ命令の実行

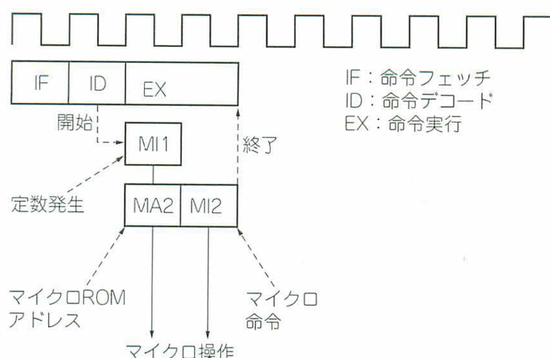


図8 1ステップ目のROMアクセスを省略

加減算のような単純な命令は，通常，1〜2ステップのマイクロ命令で実行できるので，マイクロROMをアクセスするための1クロックは性能に影響を与える．これを省略する方法がある．もともと加減算のような単純な命令は，たとえば，

(1) ソースオペランドを読み出し，演算器に演算の種類を設定する．同時にオペランドを演算器に入力する．

(2) 演算結果をデスティネーションオペランドに書き込む．

というマイクロ命令で実行できる．これは，加算であろうが減算であろうが，同じマイクロ命令で実現できるのである．つまり，マイクロ命令は固定されているので，単純な命令の場合は，最初の1ステップに関してはマイクロROMを参照することなく定数発生器でマイクロ命令を生成して実行してよい．この場合，定数発生に時間はかからないので，命令の実行を1クロック短縮できる．マイクロプログラムの2ステップ目以降は，1ステップ目からマイクロROMをアクセスしておけば，遅延なしでマイクロ命令を読み出せる(**図8**)．

● 2レベルのマイクロプログラム

マイクロプログラムを2レベルにして，マイクロプログラムの融通性を増加させたり，変更や修正を容易にすることが考えられる．マイクロROMの容量を減少させるためには，定型的な処理はサブルーチンとして1か所にまとめ，そこをコールするようにする．特に，垂直型のマイクロプログラムの場合，垂直型はエンコードが複雑になるため，サブルーチンを水平型にしてマイクロ命令のデコードの負荷を軽減することもある．実際には，マイクロROMと別個に水平型マイクロ命令の格納されたROMを用意し，ある種(複雑な)のマイクロ命令の場合は，その付随するROMから水平型マイクロ命令を読み出して，デコードし実行する．

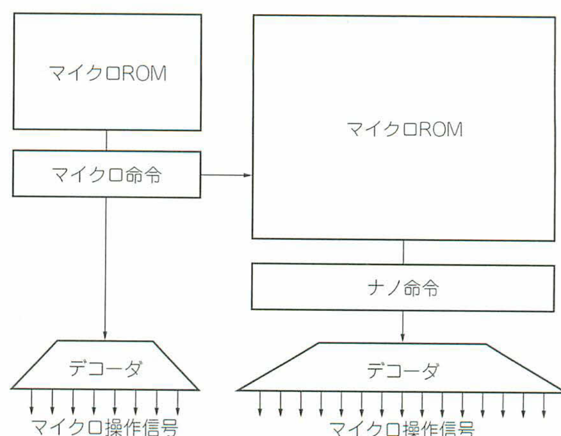


図9 ナノプログラムの概念

単純なマイクロ命令については通常のデコーダでデコードする．このような水平型のマイクロ命令は，マイクロ命令よりも細かい処理をするという意味で，ナノ命令と呼ばれる．複雑な命令や定型処理はマイクロ命令をデコードする前に，ナノROMのアドレス情報を生成するのである．**図9**にナノプログラムを使用する場合の構成図を示す．

単純な構成なら，一つのマイクロ命令は一つのナノ命令に対応する．この場合では，実質，水平型のマイクロプログラムと大差はない．ただし，ナノROMの容量を削減するため，単純なマイクロ命令はそのままデコードして実行する．経験的に(垂直型の)マイクロ命令のほうが，(水平型の)ナノ命令よりもビット長が短くなるからである．ナノプログラムをサブルーチン的に利用する場合は，一つのマイクロ命令に対して複数のナノ命令が対応する．つまり，マイクロ命令によってナノROMのアドレスが与えられると，そこから逐次的にナノ命令を読み出し，その処理の終了コマンドを実行するまで実行を続ける．この方式は，アルゴリズムの記述にはマイクロ命令を用い，実際の(複雑な)ハードウェア制御にはナノ命令を用いることになるので，効率的なマイクロプログラミングが行える．

聞くところによると，8086は1レベルのマイクロコード制御であるが，68000は2レベルのマイクロコード制御を行っていたという．

● マイクロプログラム制御方式の欠点？

これまで述べてきたように，マイクロプログラムはハードウェアのマイクロ操作信号を容易に生成でき，変更も容易で，保守もしやすい．しかし，現在ある

MPUでマイクロプログラムが使用されるケースは，最大手のIntel系列以外はあまり見あたらない．その理由はなぜだろうか．

それは，世の中のMPUの風潮がRISCになっているのと関係がある．そもそも，RISCは1クロックで実行できるような単純な命令しかサポートしない[注]．その動作を実現するマイクロ操作信号も比較的単純である．こうなると，MPU内部に(マイクロ)ROMをもち，(マイクロ命令の)デコーダをももつということは回路規模の増加になる．別にRISCをマイクロプログラムで制御してもかまわないのだが，それではハードウェア資源のむだ遣いである．つまり，マイクロプログラムが用いられない理由は，それに致命的な欠点があるわけではなく，必要性がないからである．

x86アーキテクチャは，互換性のために，古き良き(？)CISCの命令をいまだにサポートしている．これらの複雑な処理を行うCISC命令は，今でもマイクロプログラムによって実現されている．よく，マイクロプログラムによる実行は遅いといわれるが，これはかなり誤解を招く表現である．マイクロプログラムで実行するような複雑な命令は1クロックで処理することができないので，パイプライン処理に乱れが生じ，結果として命令のスループットが悪くなる．つまり，マイクロプログラムに原因があるのではなく，長くばらつきのある命令実行時間に問題があるのだ．

もっとも，マイクロプログラムを使用するとROMの読み出しに1クロックを必要とし，命令の最低実行時間が2クロックになるために遅いという意見もあるだろう．基本命令に関しては，**図8**に示したような方法でROMの読み出し時間をなくすこともできる．パイプライン制御を行う場合，スループットは最長のステージに律速されるためである．しかし，命令実行中は次の命令のデコード中でもあるので，命令実行ステージが始まる前にマイクロROMのアクセスを開始して，最初の1クロックの遅延を稼ぐことも可能ではある．これは，マイクロROMのアドレスがデコード後に決定することを考えると，速度的に非常に厳しい．ただし，デコードと実行ステージの間にオペランドリードステージを新設すれば実現は可能である(パイプラインのステージが増加した分，分岐の性能が悪くなるが)．

あるいは，マクロコードをタグ，そしてマイクロ命令をデータとする，マイクロプログラム用のキャッシュを内蔵する方法もあるかもしれない．まあ，マイクロプログラミング制御を行うのは主としてCSICであり，CISCでそんなにスムーズに流れるパイプラインを有するMPUはまれなのだが．このへん辺のオーバヘッドはあまり考慮されていないような気がする．結論としては，うまく作れば，マイクロプログラム制御が遅いということはない．

結局，マイクロプログラム制御で最高性能を出そうとすれば，現在の高速MPUの内部で行われているキャッシュ，パイプライン，分岐予測，投機実行といった処理をマイクロプログラムの制御に適用することになる．これは，マイクロプログラムの制御機構そのものがRISCあるいはVLIWプロセッサと同一視できるということを示す．つまり，MPUとは，マイクロコードで動作するRISC(VLIW)プロセッサが，マクロ命令を解釈し，実行するコンピュータシステムそのものということである．

もう少し歴史的に考察すれば，マイクロプログラミングの生まれた背景には，主記憶のアクセス時間の遅さがある．マイクロプログラミングが全盛だった1960～1970年代に，プログラムを格納する主記憶として磁気コアメモリが用いられていた．これは，半導体メモリ(ROM)と比べると，アクセス時間が10倍程度遅い．そこで，主記憶からフェッチするマクロ命令のコードサイズを小容量かつ多機能にし，それをアクセス時間の速いROMに格納されたマイクロプログラムで高速に実行することが考えられた．いきおい，一つの命令に種々の機能を盛り込む傾向になる．MPUの命令セットはマイクロプログラムで実行されることを前提として設計されている．これは，まさにCISCの考え方である．しかし1970年代の後半からは，キャッシュメモリというアクセス時間の速いメモリを利用することが可能になり，マクロ命令のフェッチとマイクロ命令のフェッチに時間差がなくなった．こうなると，上述したように，わざわざマイクロプログラムを用いる利点がなくなる．将来的には，キャッシュメモリに比べて非常にアクセス時間の短いROMが開発されるか，キャッシュメモリ自体が有効でなくなるような技術的革新が起きない限り，マイクロプログラミングの復権はなさそうである．

ところで，MPUが実行するマクロ命令がマイクロ

注：最近では他社との差別化のために，RISCでも複雑な命令をサポートする傾向にある．

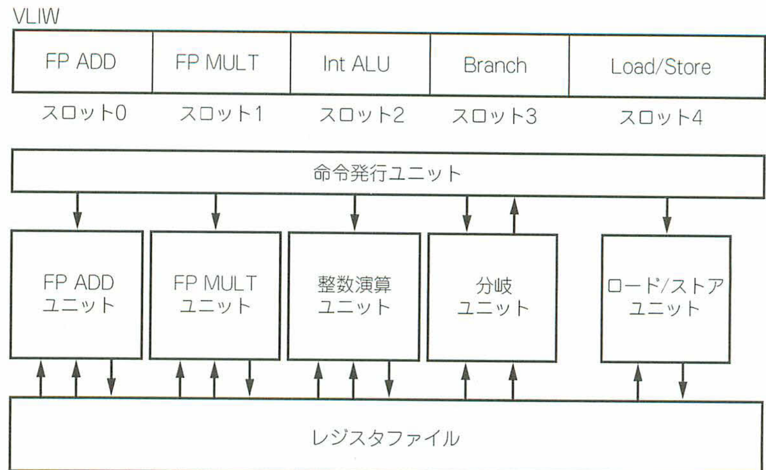


図10 VLIWの概念図

命令そのものだったらどうだろうか．その場合，プログラムの実行を最高速で，しかも並列に実行することが可能になる．事実，DSP(Digital Signal Processor)でのプログラミングはマイクロプログラムの香りを残している．プログラムはDSPの内部ユニットと密接に関連しており，プログラマはその内部状態を考慮して最適な動作になるように調節する必要がある．できるだけ多くのユニットを並列に動作させるようにするのがキーポイントである．そのため，(アセンブラによる)プログラムは非常に難しいが，これによって得られる性能は非常に高速である．世間的には「DSP＝高速プロセッサ」というのが常識であるが，これはプログラムがハードウェアをもっとも高い効率で動作させるためである．以下に述べるVLIWに関しても，そのようなイメージが付いて回る．読者の方々はどのように思われるであろうか．

2 VLIWの概念

● VLIWとは

VLIW(Very Long Instruction Word)とは，その名称のとおり「非常に長い命令語」を意味する．一般に，命令は128ビット程度の固定長で，MPUが持つ機能ユニットと1対1に対応する「スロット」という領域から構成される．スロットには対応するユニットを制御する命令が埋め込まれ，VLIWの1命令が実行されると，各スロットの命令が同時実行される．つまり，1ステップで複数の命令が実行される．**図10**にVLIWの概念図を示す．実行に際し，スロット間の依存関係は考慮しない．VLIWはスーパースカラとは異なり，ハードウェアが同時実行できる命令を自動判別するわけではない(命令内で明示的に指定されている)ので，命令発行ユニットを簡略化できる．

一方，それぞれの命令の各スロットが最大限に機能するように割り当てる必要があり，この非常に高度な技術を実現するという負荷をコンパイラに課すことになる．コンパイラは与えられた命令列から同時実行可能な命令の組を探し出し，VLIW命令の各スロットに割り当てる．スロットに入るべき命令がない場合はNOP(No Operation)を入れる．これを命令スケジューリングという(**図11**)．

スーパースカラとVLIWを比較すると，MPU内部



図11 コンパイラによる命令スケジューリング

に同時実行を行うしくみを実装する必要がなくハードウェアを簡略化できるという利点がある反面，コンパイラが頑張っても，常に最高の個数(スロットの数)の命令を同時実行できるとは限らず，NOPが余分に発生する分だけプログラムのコード効率が悪くなる(コードサイズが増加する)という欠点がある．

VLIWの発想は，Control DataのCDC6600やIBM 360/91という最初のスーパーコンピュータで採用されていたマイクロコードの並列実行方式に由来する．それらのコンピュータが活躍した1960～1970年代では，アレイプロセッサや専用のシグナルプロセッサがROMに格納されたVLIWに良く似た語長の長い命令を用い，高速フーリエ変換などのアルゴリズムを計算していた．

真のVLIWマシンは1980年代初期に三つの会社(Multiflow，Culler，Cydrome)から発表されたミニスーパーコンピュータだった．それらは商業的には成功しなかったが，これらのコンピュータに適用されたコンパイラ技術はむだにはならなかった．その後，HP(Hewlett Packard)はMultiflowを買収し，現在のHPのVLIWのコンパイラ開発はMultiflow出身のJosh FisherとCydrome出身のBob Rauを中心に行われているという．トレーススケジューリングとソフトウェアパイプライニングは，それぞれ，FisherとRauが先駆者であり，現在もVLIWコンパイラ技術の中心的役割を果たしている．なお，VLIW用に開発されたコンパイラの並列化技術の多くがスーパースカラ用のコンパイラに採用されて成功を収めているのは痛烈な皮肉である．

ところで，VLIWの嚆矢となったMultiflow-7/300は二つの整数ALU，二つの浮動小数点ALU，分岐ユニットを有していた(これらは複数のチップで構成されていた)．その256ビットの命令語は七つの32ビット長のオペレーションコードを含んでいた．各整数ユニットは130nsごとに二つのオペレーションを実行できた(合計4命令)ので，約30.8MIPSという性能になる．当時としてはかなり高性能な部類だ．また7/300を複数組み合わせて，より高性能な512ビットや1024ビット幅のマシンを構成することもできた．

一方，CydromeのコンピュータであるCydra-5は，各命令を六つの40ビットの操作として順次実行する特殊モードを備えていたが，256ビットの命令語を使用していた．そのため，そのコンパイラは並列コードと従来どおりの逐次的コードをミックスしたコードを生成したという．Cydra-5はMultiflowとは異なり，プレディケーション，ソフトウェアパイプラインといったVLIWコンパイラの基本的な並列化技術をすでに採用していたといわれている．

双方のLVIWマシンとも複数のチップで構成されていたが，最初の1チップのVLIWは1989年に発表されたIntelのi860であるといわれている．二つの32ビット命令(整数と浮動小数点)を64ビット命令とみなし，1度に命令キャッシュから取り込んで同時に実行するデュアル命令モードを備えていた．64ビット長なのでLIW(Long Instruction Word)と呼ぶのが正しいかもしれない．i860は，世間にはDSPとして受け入れられ，もっぱらグラフィックアクセラレータとして利用された(最近はRAIDコントローラ用途のほうがメジャーかも)．なお，i860では，命令を正確に実行する責任はハードウェアよりもむしろコンパイラに任せていた．結果としてi860は成功しなかったが，すべてソフトウェア任せというハードウェアの自由度の低さが一因だったのでなかろうか．結局，コンパイラがうまく開発できなかったのだろう．

ただ，当時のコンパイラ技術では，プログラムの中で並行に実行できるのは2命令程度という報告があった(5命令という報告もあったようだが)．また，先進的な並列化技術もコンピュータの非力さゆえにコンパイル時間が現実的でなく，VLIWの登場は時期尚早であったといえる．

● トレーススケジューリングとソフトウェアパイプライニング

VLIWのコンパイラの基本技術である，トレーススケジューリングとソフトウェアパイプライニングについて簡単に説明しておこう．

コンパイラは，ある命令を起点として命令列を探索し，それが分岐に突き当たり，プログラムの流れが変わるまでを基本ブロックとして最適化処理を行う．つまり，命令がループになっている場合，基本的には，異なるループ処理間では同時に最適化を行えない．しかし，(おそらくは一つのループを含む)基本ブロックをまたいで命令のスケジューリング(並び替え)を行う手法がトレーススケジューリングなのである．

ソフトウェアパイプライニングとはループの内部をパイプライン的に処理できるように並び替える手法である．それには，ループアンローリングという手法が基本となる．これはループをアンロール(部分的に展開)するものである．たとえば，

```
for(i=1;i<=N;i++){
      命令1
      命令2
      命令3
}
```

というループを考える．これを，アウトオブオーダなスーパースカラで実行する場合，命令1，命令2，命令3の3命令では依存性が高くてうまく並列実行できないことがある．その場合たとえば，2ループを一つの単位として，

```
for(i=1;i<=(N/2);i+=2){
      命令1
      命令2
      命令3
      命令1
      命令2
      命令3
}
```

とアンロールしてみれば，並列実行できる組み合わせが見つかるかもしれない．しかし，インオーダなパイプラインではあいかわらず依存性は解消されていない．そこで，ソフトウェア(コンパイラ)で，並列実行できるように命令のスケジューリング(並び替え)を行うことが考えられる．たとえば，

```
for(i=1;i<=(N/2);i+=2){
      (命令1 ; 命令1)
      (命令2 ; 命令2)
      (命令3 ; 命令3)
}
```

のようにスケジューリングすればカッコ内の組が並列に実行できるとしたら，処理時間が半分になる．このようにアンローリングしたループ内の命令の並び替えをソフトウェアパイプライニングという．パイプラインというイメージからは，

```
for(i=1;i<=(N/2);i+=2){
      (命令1 ;      )
      (命令2 ; 命令1)
      (命令3 ; 命令2)
      (      ; 命令3)
}
```

のように，命令がオーバラップする感覚で，並び替えられることを想定しているのかもしれない．これは，MPU内部のハードウェア資源をむだなく使い回せるような並びになるはずである．

3 VLIWの実際(1)─Itanium

VLIWの実現性を疑問視する声が多いのは事実であるが，最近では珍しく成功した(と現時点では記しておこう)二つのMPUがある．IntelのItanium(第一弾のコードネームはMerced)とTransmetaのCrusoeである．どちらもx86命令を実行する方式としてVLIWアーキテクチャを採用している点が興味深い．これらの特徴を見ていこう．

VLIWプロセッサを商品化しているメーカは実際には，結構ある．たとえば，富士通のFR-Vは自社のサイトで大々的に宣伝している．しかし，これらが汎用MPUとして普及しているという話は聞かない．所詮は，組み込み制御用途のDSPといった感覚なのだろう．

また，VLIWは，最近流行のリコンフィギュアラブルプロセッサ(これは多くの場合，DSPの代用である)の制御用プロセッサとして利用されることが多い．これは，VLIWがDSPの制御にとって相性がいいという理由もあると思われる．

● 開発の背景

VLIWはHPの技術によって支えられているといっても過言ではない．事実，HPはPA-RSICの最新機種として，PA-9000というVLIWマシンを1998年に発売すると発表していた．しかし，1994年，HPはIntelと共同で64ビットMPUであるIA-64の開発を行うことを表明した．それと同時にPA-9000の開発を凍結した．IA-64はEPIC(Explicitly Parallel Instruction Computing)と呼ばれるVLIWライクな命令を主体とする新しい概念を採用している．EPICは命令セットアーキテクチャと同時に高性能化などのインプリメンテーションを同時に定義する．といっているが，単なるマーケティング戦略にすぎない．EPICも明らかにVLIWである．

x86とIA-64の主な特徴の比較を**表1**に示す．64ビットMPUとしては後発になるため，新しい技術を提供することが必要であり，それにより明るい未来が約束されているように思えた．しかし共同開発とは名ばかりで，実質的なIA-64の開発はIntelによって行われた．当初の計画とは異なり，PA-RISCとの互換性はなくなり，IA-64とx86のみのサポートになった(妥協案としてダイナミックトランスレーションというエミュレーション技術が提案されてはいるが)時点で，

表1　x86とIA-64の比較

	x86	IA-64
命令形式	複雑で可変長の命令を1度に1命令処理する	単純で固定長の命令三つを一つのグループにバンドルして同時に処理する
実行順序	命令列の並び替えと最適化を実行時に行う	命令列の並び替えと最適化をコンパイル時に行う
分岐予測	予測した分岐先を投機的に実行する	分岐先と分岐元の両方を投機的に実行し，不要な側を無視する
メモリ参照	必要になったときにデータをメモリからロードする．キャッシュを最初に参照	必要になる前に投機的にデータをロードする．同じくキャッシュを最初に参照

HPは共同開発から手を引いた感がある．

実際，HPは1999年6月からPA-RISCの最新機種であるPA-8500の出荷を開始した．同時に，PA-8600/8700/8800/8900というロードマップを発表してPA-RISCの存続をアピールした．PA-8500を使用したサーバであるHP9000はIA-64にアップグレード可能とはなっているが，それはIA-64の最初のItanium(Merced)ではなく，次機種のMcKinley(コードネーム)以降であるとしている．HPのいいぶんは「Itaniumの性能は期待外れ」ということらしい．性能問題が尾を引いたのか，Itaniumの開発は遅れに遅れ，2000年に発売に漕ぎ着けたものの，その年の7月にはさらなる改版が必要として，2001年の前半まで発売を延期した．発表された動作周波数も800MHzと予想をはるかに下回っている．その時点でのHPの公式見解は「いまだにIA-64に期待する」というものであるが，Itaniumの失態が繰り返されるようではHPの離脱もありえる．また，SGIなども，Itanium搭載のLinuxマシンを発表していたが,いまひとつ盛り上がりに欠けていた．

2002年にItanium2(McKinley)がリリースされてもその低調さに変わりはなかった．しかし，HPは唯一，そのプロモーションに積極的だった．Itanium2になって，やっと望みどおりの性能が達成されたということなのだろう．それまでの動向を見ていると，Itanium2の成功はHPの頑張りにかかっていたといっても過言ではない．

しかし，2003年6月に2代目Itanium2(Madison)が発表されると事情が少し変わってきた．Madisonは，L3キャッシュの容量が増えたことにより，McKinleyよりも30～50％の性能向上となり，やっと競合他社を明確に凌駕する性能を達成できた．これをきっかけに，HP以外での採用も増加し，Itaniumの快進撃が始まる兆しがみえてきた．

ItaniumはIA-32の命令セットを直接実行する専用のユニットを持つ．Itanium2のホワイトペーパーによれば，サポートする命令はKatmai(PentiumⅢ)レベルであり，性能は300MHzのPentiumProと同程度となっている．実際に使用したユーザーの感想によると，Pentiumの75MHz程度の体感性能で想像を絶する遅さともいわれている．これでは，当初の予定と異なり，IA-32の性能は期待できそうにもない．バイナリ変換を行わなくても一応動作するという宣伝文句以上の意味はないと思われる．当のIntel自身ですらその使用を推奨していない．

2003年4月24日，IntelがItanium用のx86(IA-32)エミュレータを開発中であることが明らかになった．これは，CNETのスクープであり，このエミュレータは「IA-32 Execution Layer(IA32-EL)」(コードネームはbtrans)と呼ばれる，OSと連係してIA-32プロセッサをエミュレートするソフトウェアである．具体的には，Xeon(IA-32)命令からItanium2命令へのトランスレータである．このIA32-ELを使えば，1.5GHz動作のItanium2は1.5GHz動作のXeon MPとほぼ同じ速度で32ビットコードを稼働させられるという．

このIA32-ELは2004年下半期にリリース予定のWindows Server 2003のService Pack 1に組み込まれる予定だという．しかし，最近の報道では，1.5GHz動作のItanium2で1.4GHz動作のXeon相当とその性能が下がっているのが気になる．

この噂を打ち消すように，2004年1月23日に日本で開催されたIntelの事業展開の説明会では，IA32-ELの性能はムーアの法則にしたがって進化し，将来的にはItaniumのハードウェアエミュレーション機能は削除する方向との説明がされた．2008年にはIA32-ELがIA-32(Pentium4)の性能を凌駕すると予想するアナリストもいる．

AMDのOpteronは64ビットコードだけでなく32ビットコードも効率的に実行できるので，これと比べるとItaniumが見劣りするのは明らかであり，その対策と思われる．IA32-ELにより，32ビットのソフトウェア資産をItaniumにも流用できるようになり，それが理由でItanium採用に消極的だった顧客に対して優位性をアピールすることができる．

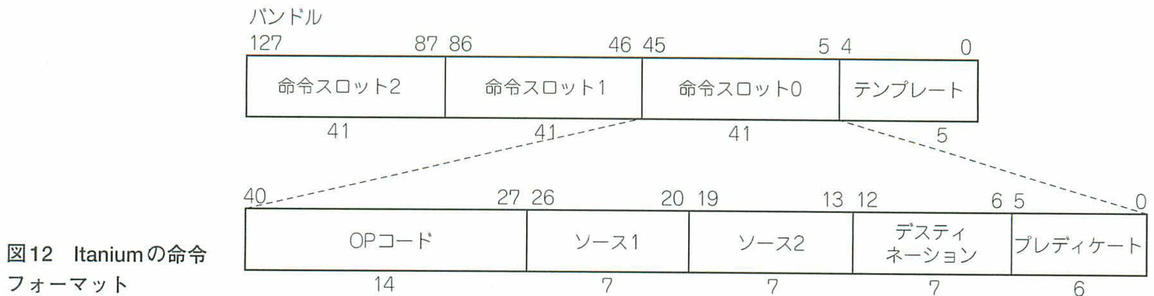


図12　Itaniumの命令フォーマット

● 命令フォーマット

Itaniumの命令フォーマットを**図12**に示す．41ビットの3オペランド命令が3個と，5ビットのテンプレートを組み合わせた128ビット長の命令である．これはバンドル(抱き合わせ)と呼ばれる．

各スロットに格納される命令のプレディケートとは「述語」という意味で，命令の実行条件を示す．分岐属性とも訳される．Itaniumは64本の1ビット長のプレディケートレジスタを備え，各レジスタは「真」または「偽」の情報を保持する．命令中のプレディケート領域はプレディケートレジスタの番号を表し，そのレジスタが「真」ならば命令を実行する．これはRISCで採用されている条件MOVEの発展形で，条件分岐での場合分けを(性能が低下する)分岐命令を使わずに表現できる．また，各命令は3オペランド命令である．

レジスタは128本用意されているが，常時使用できるレジスタはR0～R31の32本のみである．R32～R127は`alloc`という命令でスタックフレーム領域として定義すると使用可能になる．サブルーチンコール先で確保されたスタックフレームはサブルーチンからリターンすると自動的に消滅する．このようなレジスタ構造をIA-64ではスタックレジスタと呼んでいる．

バンドルに含まれるテンプレートは三つのスロットの命令の種類を指定する領域である．これは内蔵する演算器と密接に関連し，I(整数)，M(メモリ)，F(浮動小数点)，B(分岐)の中から指定できる．テンプレートは同時に並列実行できる命令の区切りも指定する．ハードウェア資源(演算器の数)が十分にあれば，基本的に，すべての命令を並列実行できるが，実装する演算器の数に依存して区切りが決定される．これは5ビットの領域なので32種類の組み合わせを指定できるのだが，Itaniumでは**図13**に示す24種類が定義されている．

なお，バンドルは1対1でPA-RISCの命令に変換可能というが，実際はどうなのだろう．

テンプレート	スロット0	スロット1	スロット2
00	M	I	I
01	M	I	I
02	M	I	I
03	M	I	I
04	M	I (L+X)	
05	M	I (L+X)	
08	M	M	I
09	M	M	I
0A	M	M	I
0B	M	M	I
0C	M	F	I
0D	M	F	I
0E	M	M	F
0F	M	M	F
10	M	I	B
11	M	I	B
12	M	B	B
13	M	B	B
16	B	B	B
17	B	B	B
18	M	M	B
19	M	M	B
1C	M	F	B
1C	M	F	B

▮並列実行できる命令の区切り

図13　バンドルの組み合わせ

● 内部構造

図14にItaniumのブロック図を示す．演算器の構成は，分岐ユニット3個，整数/メモリユニット4個，浮動小数点ユニット2個である．二つのバンドルが同時に処理されるので6命令の同時発行が可能である．また，L1，L2キャッシュを内蔵し，外部にはL3キャッシュが接続される．

Itaniumがもっとも効率良く命令の並列実行を行うためには，コンパイラによるヒント(制御情報)を命令に反映させることが肝要である．VLIWではコンパイラによるいろいろな並列化技術が研究されているが，

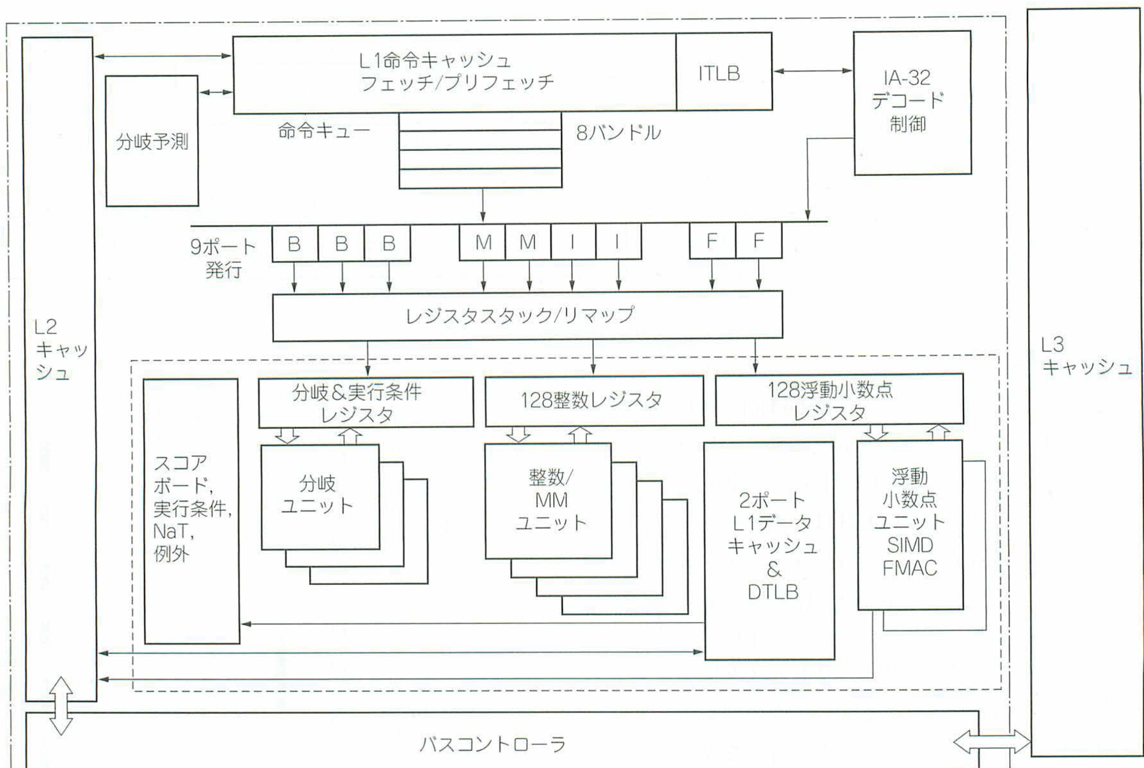


図14　Itaniumのブロック図

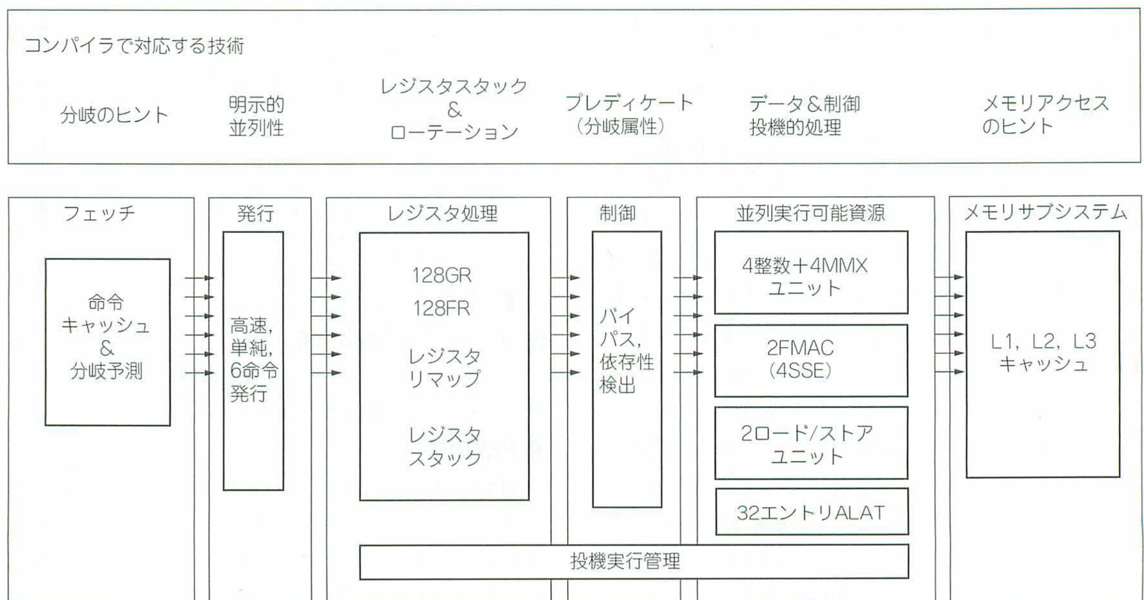


図15　Itaniumのハードウェアとコンパイラの技術

それらをいかに適用できるかでMPUの性能が決まるといっても過言ではない．**図15**に各ハードウェアブロックに対してどのようなコンパイラ技術が必要になるのか，その一例を示す．

● パイプライン

図16(a)にItaniumのパイプラインを示す．10ステージからなるインオーダなパイプラインである．パイプラインは，フロントエンド，命令供給，オペラン

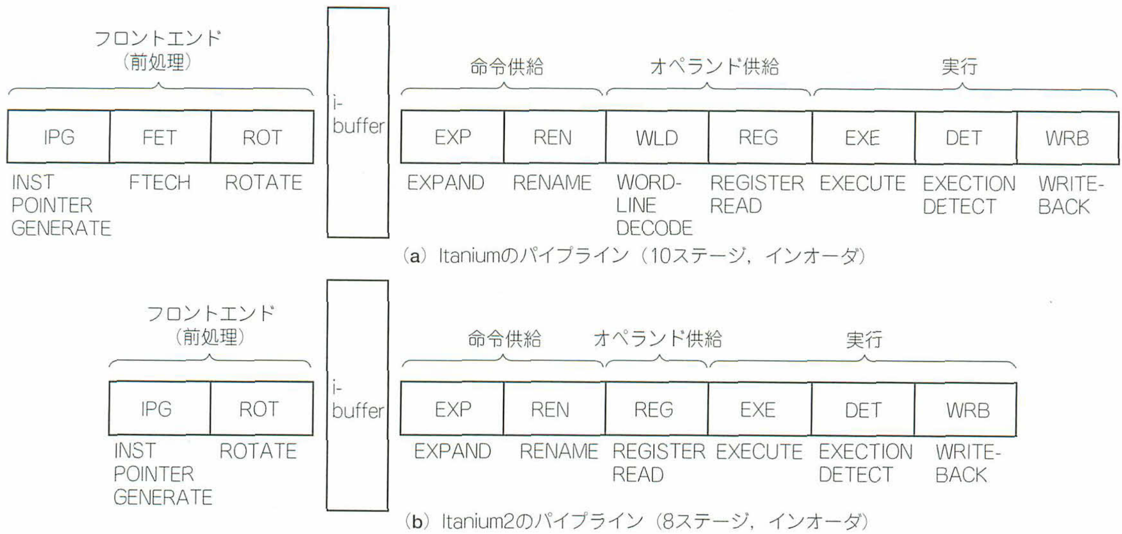


図16 ItaniumとItanium2のパイプライン

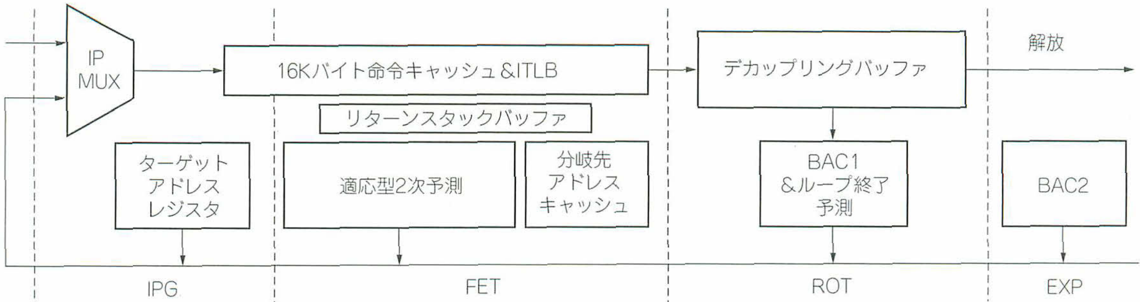


図17 Itaniumのフロントエンド

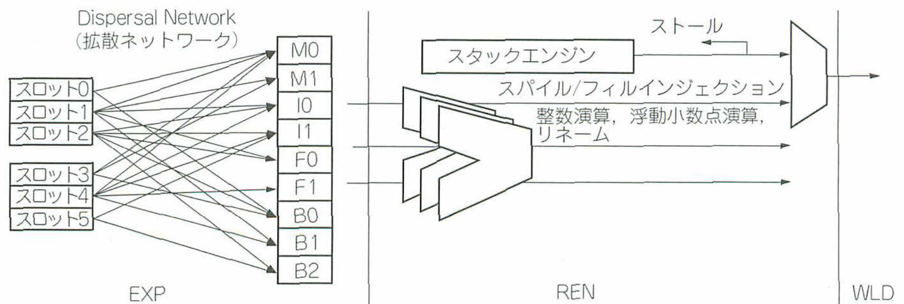


図18
Itaniumの命令供給段階

ド供給，実行の段階に大別される．

フロントエンド(**図17**)の段階では，1サイクルに最大6命令(2バンドル)をプリフェッチまたはフェッチし，8エントリのデカップリングバッファ(命令キュー)に格納する．分岐予測もここで行う．

命令供給の段階(**図18**)では，最大6命令を九つのポートに振り分ける．これはDispersal Network(拡散ネットワークとでも訳すのか)によって行われる．将来的には各スロットをすべてのポートに振り分け可能になる予定であるが，Itaniumではスロットとポートの対応に制限がある．レジスタのリネームやスタック操作(動的なレジスタ割り当て=Spill/Fill)もここで行う．

オペランド供給の段階(**図19**の左側)では，レジスタリードとバイパス(フォワーディング)処理を行う．その他，オペランドの依存性のチェック，分岐属性の依存性チェックを行う．実行の段階(**図19**の右側)では，四つの1サイクルレイテンシのALU，二つの

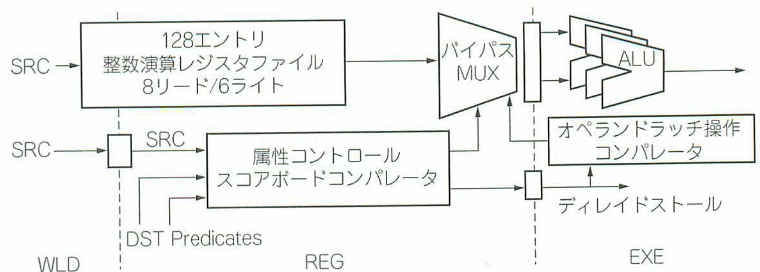


図19 Itaniumのオペランド供給と実行段階

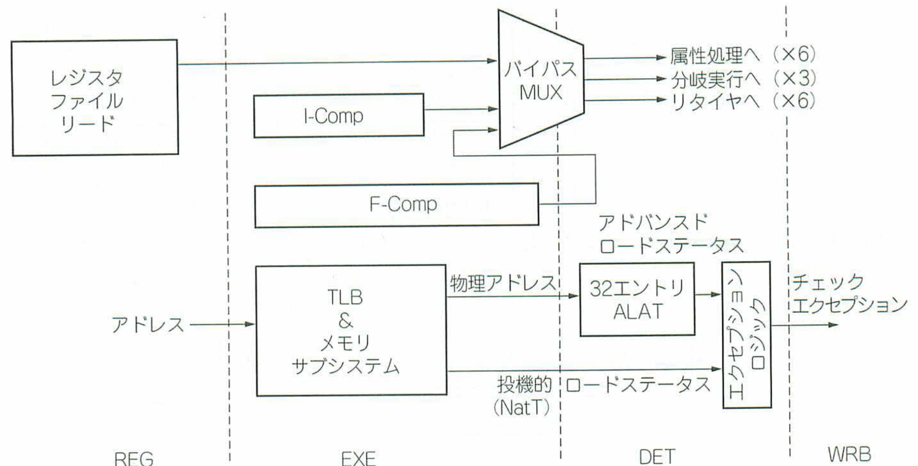


図20 Itaniumの例外検出

ロード/ストアユニットで命令を実行する．また，投機ロードの制御，分岐属性の処理，例外検出，リタイア処理(**図20**)を行う．

● Itanium2での性能改善

Itanium2(McKinley)ではItaniumに比べてMHz当たりの整数性能(SPECint2000)が25％以上向上した．これは当初の予定であるItaniumの1.5〜2倍の性能をほぼ実現している．これらは，動作クロックの高速化，バスの高速化，マイクロアーキテクチャの改良，3MバイトのL3キャッシュの内蔵(Mercedでは最大4Mバイトまでを外付け可能)による恩恵である．

Itanium2では，1クロックに発行できる二つのバンドルの組み合わせを増やすことに重点が置かれている．そのために演算器の数を増やした．二つのバンドルは，最大4個のメモリ演算(M)，最大6個のALU演算(M/I)，最大4個の整数演算(I)，最大6個の分岐(B)，最大2個の浮動小数点演算(F)を含むことができる．

しかし，Itaniumでは演算器の数が少ないため，バンドルの組み合わせが制限されていた．Itanium2ではほとんどすべての組み合わせを可能にしている．このため整数ユニット(ALUを含む)は4個から6個に，ロード/ストアユニットが2個から4個，マルチメディアユニットが4個から6個に増加された．浮動小数点ユニットは4個，IA-32ユニットは1個と従来どおりである．逆に浮動小数点のSIMDユニットは4個から2個に減少された．これは，SIMD命令はサーバ機での必要性が少ないためであろう．

その結果，Itanium2で同時に発行可能なバンドルの組み合わせは**表2**のようになった．つまり，Itanium2では組み合わせ可能な組み合わせの75％をサポートする．Itaniumでは28％だった．

図16(b)にItanium2のパイプラインを示す．Itaniumと比べて，FET(FETCH)ステージとWLD(WORD LINE DECODE)ステージが削除され，Itaniumの10ステージから8ステージに減少している．その主な目的は分岐命令の高速化(分岐予測ミス時のペナルティを減少させる)にある．

FETステージは命令キャッシュを参照するステージであるが，キャッシュ回路が最適化され，IPG(命令ポインタの生成)とFETが同時に実行されるようになった．WLD(レジスタファイルのワード線を決定)はレジスタファイルの設計を見直すことで不要になった．

表1　x86とIA-64の比較

		発行可能な第2命令組									
		MII	MLX	MMI	MFI	MMF	MIB	MBB	BBB	MMB	MFB
発行可能な第1命令組	MII	◎		◎	◎	◎	◎	○	○	◎	○
	MLX		○	◎	○	◎	○	○	○	◎	○
	MMI	◎	◎	◎	◎	◎	◎	◎	○	◎	◎
	MFI	◎	○	◎	○	◎	○	○	○	◎	○
	MMF	◎	◎	◎	◎	◎	◎	◎	○	◎	◎
	MIB	◎	○	◎	○	◎	○	○		◎	○
	MBB										
	BBB										
	MMB	◎	◎	◎	◎	◎	◎	◎		◎	◎
	MFB	○	○	◎	○	◎	○	○		◎	○

1）第1命令組では分岐命令を発行できない
2）◎はItanium2のみで発行可能，○はItaniumとItanium2で発行可能

Itanium2では分岐命令の処理自体も高速化されている．命令キャッシュの各ライン内に，命令(2個のバンドル)だけでなく，分岐予測情報と分岐先アドレスをも格納することで，ペナルティなしで次の命令フェッチ先を生成できる．この機構は「0-cycle branch re-steer」と呼ばれている．

加えて，分岐予測機構も次のように強化されている．

(1) 分岐履歴テーブル

L1キャッシュとデコーダの2レベルで行われ，第2レベルでは4ビット×12Kエントリのテーブルをもつ．

(2) パターン履歴テーブル：16K個の2ビットカウンタ

(3) リターンスタックバッファ：8エントリ

(4) 間接ターゲット予測：8個の分岐レジスタを使用

(5) 完全なループ予測：ループの終了を計算して予測

さらに，命令のプリフェッチ機能も強化されている．これは，基本的には，コンパイラによって指定される．

(1) デマンドプリフェッチ

次の命令をL2キャッシュにヒットする限り，L1キャッシュにプリフェッチする．

(2) ストリーミングプリフェッチ

コンパイラはすべての分岐命令に，プリフェッチを行うためのヒント情報を付加することができる．最大四つのプリフェッチ要求を処理することができ，それは停止条件が成立するまで継続される．

(3) ヒントプリフェッチ

コンパイラが使用できる2種類の専用命令がある．分岐予測(brp)と分岐レジスタへの転送(movbr)である．これらの命令は16個のバンドルをプリフェッチできる．

図21にItanium2の各パイプラインステージにおけるデータ処理のようすを示す．

● Itanium路線の進化

種々の悪評にもめげず，Itanium路線は堅実な進化を遂げている．そして，Intelは2003年6月30日のMadison発表で自信を得たのか，かなり強気なロードマップを発表している(**図22**)．コンピュータメーカにとって嬉しいのは将来に渡るロードマップがしっかりしていることであろう．

▶ McKinley

McKinleyの詳細が示されたのは2001年8月のIDF(Intel Developer Forum)である．そのときの説明によると，McKinleyはItanium(Merced)の1.5～2倍の性能を実現するという．具体的には，動作クロックの高速化，バスの高速化，マイクロアーキテクチャの改良，3MバイトのL3キャッシュの搭載である．

Itaniumのバスは64ビット幅で266MHz動作であるが，McKinleyのバスは128ビット幅で400MHz動作である．つまり，バスのバンド幅は3倍になる．IDFではMcKinleyの動作周波数は明らかにされなかった．結局は，Itaniumの800MHzを少し超えた900MHzで登場し，最終的には1GHzに達した．

McKinleyのL3キャッシュサイズは1.5～3Mバイトで，Itaniumの2～4Mバイトよりも小さい．ただし，ItaniumのL3キャッシュは外付けである．McKinleyではL3キャッシュを1チップに統合するため，データの高速転送が可能になる．結果として性能は向上する．

2002年4月25日，IntelはMcKinleyの正式名を「Itanium2」に決定したと発表した．これは，Itaniumをブランド化し，ハイエンドエンタープライズコンピューティングにおける新たな能力を示す象徴にしたいという希望のあらわれである．

図21　Itanium2の各パイプラインステージにおけるデータ処理のようす

結果として，McKinleyはそれなりの評価を得たが，Itanium2が爆発的に採用されるようになるのは次機種のMadisonからである．

▶Madison

このMadisonも2001年8月のIDFで紹介されている．MadisonはMcKinleyの1年後に登場した．これは，最大6MバイトのL2キャッシュを搭載し，動作周波数もさらに向上されていた．結局は1.3GHz～1.5 GHzで登場した．

2002年6月19日，Intelは，Madisonが複数のOS上で問題なく動作したと公表した．同チップは2003年に量産が始まる予定とし，Itaniumは向こう10年間使

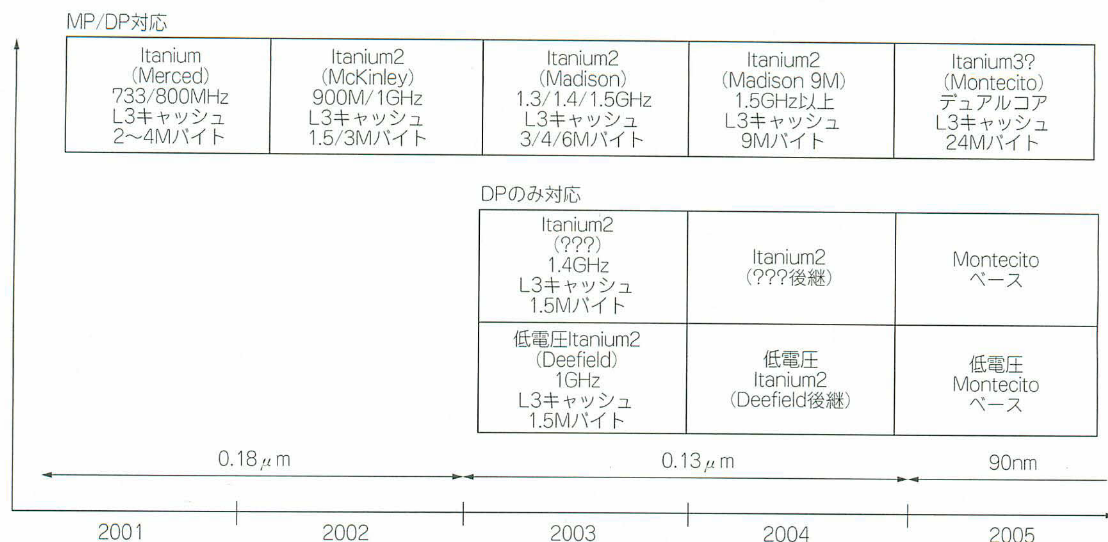


図22 Itaniumのロードマップ(2003年11月時点)

われるエンタープライズアーキテクチャになるだろうと強気である(そういえば，Pentium4に採用されたNetBurstアーキテクチャも10年の寿命を見込んでいる)．

それと同時に，Intelの社長であるPaul Otellini氏は，AMDと同様なx86-64を実行可能なMPUであるYamhillの存在を否定した．情報筋は，Itaniumの売り上げが低迷すればYamhillの登場になると予定している．Intelは32ビットMPUであるPentiumとXeonを拡張して64ビットコードを処理するための特許を保有しているらしい．

MadisonがItanium2のブランド名を継承することは，2002年10月のIDFにおいて正式に発表された．これは，MadisonがMcKinleyとそのまま差し替えられるため，同じ名称になったという．まあ，MadisonはMcKinleyのL2キャッシュの容量を増やして動作周波数を上げただけなので，当然といえば当然である．Madisonの性能は，SAP APOというベンチマークで，McKinleyの28%増ということである．

ISSCC2003で，IntelはMadisonの概要を発表した．最大の特徴は，24セットアソシアティブ構成のL3キャッシュを6Mバイト内蔵することである．チップサイズは374mm^2(約19mm角)ということなので当初の予定よりもかなり大きい．トランジスタ数はItenium2の約2倍の4億1000万になるという．動作周波数は1.5GHzで，消費電力は130Wである．

2003年6月30日にMadisonは正式に発表された．ほぼ事前のリークどおりの仕様である．

L3キャッシュの容量が増えたことにより，Madisonは，McKinleyよりも30～50%の性能向上となる．

2003年11月時点では，高性能サーバならHPのItainum2(Madison)搭載機という常識ができつつある．たとえば，HPのSuperdomeはTPC-Cベンチマークで初めて100万という値を突破した．HPとIntelの悲願がやっと達成され始めたという感じだろうか．ちなみに，Itanium2以外では，IBMのPower4を搭載したサーバがTPC-Cベンチマークの上位を占めている．

▶Madison 9M

2003年1月，IntelはItaniumのロードマップに関して発表(リーク)を行った．そこで，これまでロードマップになかった「Madison 9M」が加わった．これはMadisonの9MバイトL3キャッシュ搭載版である．Montecitoをデュアルコア化することによる日程遅れを補う中継的なプロセッサである．

▶Deerfield

Deerfieldも2001年8月のIDFで初めて紹介された．

Deerfieldの基本的なアーキテクチャはMadisonやMcKinleyと同じだが，消費電力と発熱量が抑えられているという，いわば廉価版である．現行のItaniumの消費電力が120Wであるのに対し，Deerfieldでは70～80Wになるらしい．これはPentium4と同程度である．

当初，MadisonのL3キャッシュが3Mバイトで1.3GHz版がDeerfieldとの噂もあったが，2003年6月30日のMadisonの発表では別物であることが判明した．

同時に発表されたロードマップではDPのラインに

は1GHz動作の低電圧版と1.4GHz動作の通常電圧版(コードネーム不明)があり，どちらも1.5Mバイトの L3キャッシュを持つ．1GHz動作の低電圧版がDeerfieldである．1.4GHz動作版はこの時点で新たにロードマップに付け加わった製品である．

2003年9月8日，Deerfieldが正式発表された．1GHz動作で1.5MバイトのL3キャッシュを搭載するLow Voltage Itanium2がそれである．

従来のItanium2の最大電力が130Wだったのに対し，Deerfieldの消費電力は62Wと半分以下であり，処理能力はMcKinleyと同等であるという．

また同時に，1.4GHz動作で1.5MバイトのL3キャッシュを搭載する省機能版が発表された．これらは事前の予想どおりである．コスト，設置場所，消費電力量を重要視し，クラスタ結合して高性能コンピュータシステムを構築できるようになっている．1.4GHz版の価格は1172ドルである．

▶Montecito

2002年1月末に，業界の噂としてMontecitoの存在が明らかになった．

Montecitoは2004年から2006年にMadisonの後継として登場し，2001年にCompaqから獲得したAlphaの技術が適用されるという．その時点では，どのような技術かは不明だったが，噂ではHyper-Threadingをサポートするという予測が大勢であった．しかし，当のIntelからの見解はなかった．

2003年1月のItaniumのロードマップでは，当初シングルコアであったMontecitoをキャンセルし，新たに，デュアルコア版のMontecitoを当初の計画より早く，2005年にリリースするという．本来なら，マルチコアは2007年の予定だった．

なお，Madison，Madison 9M，Deerfieldは130nmプロセスで製造されるが，Montecitoは90nmプロセスで製造される．また，これらのプロセッサはすべてItanium2のブランド名になり，ピン互換になるらしい．

Montecitoで特徴的なことは，各MPUコアがそれぞれ独立して18Mバイト以上のL3キャッシュを内蔵するという点である．

Column 1 Itaniumに関する個人的感想

Itaniumのパイプラインは単純でインオーダな10ステージ構成である．その性能のキーポイントは二つのバンドル(6命令)をいかに並列実行できるかにかかっている．しかし，それはコンパイラでのスケジューリングに大きく左右される．そもそも，通常のアプリケーションプログラムで六つの命令を並列に実行できるほど命令間の依存度の低い場合がありうるのだろうか．トレーススケジューリングやソフトウェアパイプライニングなどの技法でそれなりに並列度を上げることができたとしても，実際のところはどうだろうか．まあ，将来的に，コンパイラの技術が進歩すれば進歩するほど性能が上がる構成になっていると考えれば，先見の明ということもできるのだが….

Intelは2003年に入るとItaniumの大々的なプロモーションを行っている．Mercedの開発ではいろいろとトラブルが絶えなかったItaniumも，順調にデベロッパやハードウェアメーカの関心を高めているようだ．しかし，命令レベルでの並列度を上げるためにVLIW(EPIC)を採用したのに，結局は(Montecito以降)デュアルコアでスレッドレベルでの並列度向上に逃げた方針変更(？)はEPICの失敗を認めるものであるという見方もある．

VLIWとはMPUのハードウェア資源を常に動作させるようにコンパイラが最適化を行う技術であるから，ハードウェア資源の空きを自動検出して自動的に処理を割り当てるマルチスレッドやハードウェア資源を浪費するマルチコアの適用は，VLIWの利点を否定することである．

その意味でItaniumのアーキテクチャ自体が迷走しているといえなくもない．Intelの総力をもってすれば誤った選択も容易に軌道修正がきくということか．何にせよ，勝てば官軍である．

しかし，とりあえずはItaium2(McKinley)でマイクロアーキテクチャを見直し，L3キャッシュをオンダイに内蔵した．これによって，Itaniumの1.5～2倍の性能になった．さらに次機種のMadisonでは，動作周波数を向上させるとともにL3キャッシュの容量を6Mバイトに倍増したことで，性能をMcKinleyの1.3～1.5倍に向上させた．これにより，Itanium2は競合他社の性能を凌駕することになり，やっと将来への道筋が見えてきた．

しかし，American Technology Researchという調査会社のレポートによると，Intelは2004年の中旬にYamhill(x86の64ビット拡張)を実装するチップ(Prescott?)を発表し，2005年に量産に入るという．もし，これが本当ならItaniumの64ビットEWSとしての価値は暴落し，単なる科学技術計算器になり果ててしまう．2003年12月時点でItaniumの出荷台数は後発の64ビット版x86であるOpteronに抜かれたという風評もあり，Itaniumの先行きが案じられる．

▶Chivano

Chivanoも，2002年1月末に業界の噂として，Montecitoと一緒にその存在をささやかれ始めたものだ．

Chivanoの登場時期はMontecitoと同時期であり，こちらはMPUのCMP(Chip Multi-Processor)になるという噂だった．

2002年10月のIDFにおいて，Intelは2005年までにデュアルコア(CMP)のItaniumを開発する計画があることを公表した．Chivanoの噂は本当だったわけだ．Hyper-Threadingに関しての言及はなかった．これらのItaniumは90nmまたは65nmの製造プロセスで実現される予定である．これは，複数CPU構成が常識であるサーバ機において，1MPU当たりの価格を下げることに貢献する．

ChivanoはMadison正式発表時のロードマップでは消滅した．これは，シングルコアのMontecito開発をキャンセルしたことにより，Chivanoとして開発していたプロセッサを新たにMontecitoと命名し直したということであろう．

▶Tukwila(Tanglewood)

ZDNetは2003年8月，Intelから近い筋の話として，2003年9月のIDFでコードネーム「Tanglewood」と呼ばれる新しいハイエンド版Itaniumの計画が発表される見込みと報道した．Tanglewoodでは一つのシリコン上に，最大16個のプロセッサが搭載される．それでいて，消費電力は現行のItaniumと同じだという．出荷時期は未定だが，早ければ2006年になるという．

しかし，2003年9月のIDFでは，Tanglewoodは2MPUよりもずっと多いマルチコアになると述べられただけだった．ただ，TanglewoodはDEC出身のAlpha設計チームが開発しており，性能は少なくとも現在のMadisonの7倍ということなので4～8MPUというのが妥当な線ではなかろうか．

なお，2003年12月，IntelはTanglewood Music Centerとの商標紛争を避ける目的で，Tanglewoodというコードネームをtukwilaに変更した．

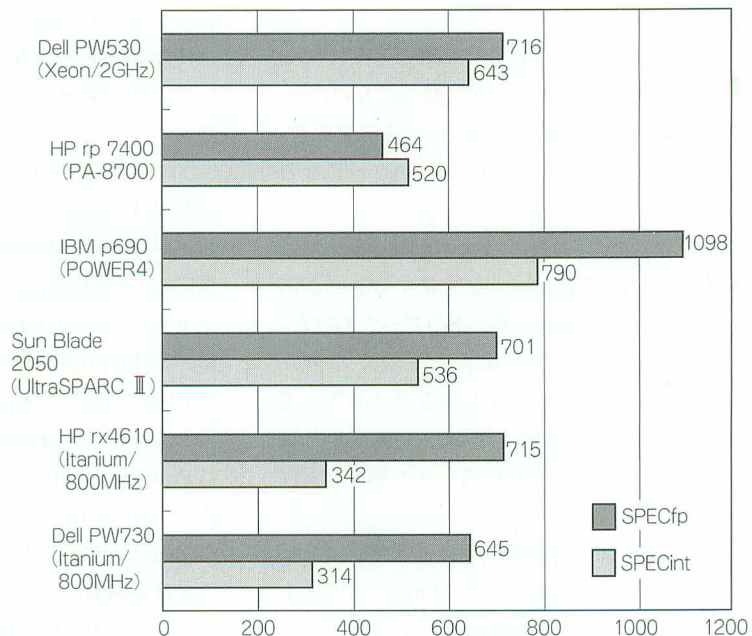
● 性能比較

さて，Itaniumは最初733MHz，その後800MHz品がリリースされたが，それを使用したEWSの性能は他社製のEWSと比べてどの程度違うのだろうか．**図23(a)**に，2001年末時点でのEWSの性能をSPEC CPU2000ベンチマークで比較する．Itaniumは，浮動小数点性能は他社並みであるが，整数性能は他社の半分ほどである．これでは，McKinleyで2倍の性能になっても，その頃には他社製品も性能向上しているはずなので，とても追いつかない．これを見る限り，EPIC(VLIWアプローチ)が成功しているとはいいがたい．

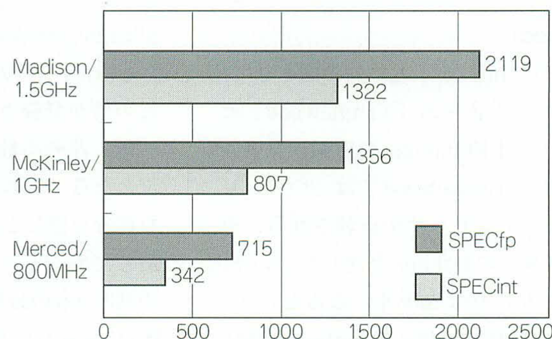
2002年5月29日，ドイツでのIDFでItanium2の性能が公式発表された．Itaniumに比べて1.5～2倍の性能というのは当初の予定通りである．発表では8MPU構成のUltraSrapc Ⅲを強く意識し，各種ベンチマークで1.5～4倍の性能を発揮するという．SPEC2000では，SPECint2000が700以上(base)，SPECfp2000が1350(base)と発表された．これが登録されると，SPECintではPower4の804に次ぐ第2位，SPECfpではPower4の1202を凌ぐ第1位となる(2002年5月時点)．IDFでは「優れた性能」と自画自賛だったが，SPECintが2GHzのPentium4と同程度の性能ということは意図的に(？)伏せられている．

2002年7月8日，IntelはItanium2の出荷を開始したと発表した．そのとき，その性能も改めて公式発表の運びとなった．HPのrx5670(1GHz)の性能だが，SPECint2000が810(base)，SPECfp2000が1356(base)となって，5月の発表よりも若干性能が向上している．この値が登録されると，64ビットサーバでは，整数性能，浮動小数点性能ともに，Power4を抜いて第1位となる．この意味で，整数性能の向上は多少意図的な部分を感じるが，コンパイラの改良によるものと理解しておこう．ただ，Itanium2搭載製品は共同開発のHPのもの以外は，日本のメーカ(NEC，日立製作所)によるものだけで，富士通とIBMは年末を目処にという発表に留まり，決して好調とはいえない．特に，Intel製品ではトップシェアのDELLが模様ながめなのが象徴的である．

米国の調査会社であるGartnerは，2002年8月に，64ビットサーバ市場においてIntelのItaniumチップは徐々に勢力(売り上げ)を伸ばしていく見通しだが，2007年でもSUNのUltraSPARCやIBMのPower搭載機を追い越せないという予測を発表した．その理由は，サーバ市場の低迷にも原因があるという．IT関連予算の締め付けが厳しく，顧客が新しい大型マシンの導入に消極的になっているという．Microprocessor Reportの主任編集者は，Itaniumシリーズの出荷量が大きく増加するのは，2003年にMadisonやDeerfieldが登場してからになると予測する．以前は，McKinley(Itanium2)の登場でItaniumシリーズの売り上げが増大すると予測されていたので，先送りされた格好で



(a) 各社のMPUとの比較



(b) Itaniumシリーズでの比較

図23　SPEC CPU2000ベンチマーク

ある．

2003年6月30日に発表されたMadisonのベンチマーク性能では，SPECint2000が1322(base)，SPECfp2000が2119(base)と発表されており，これが登録されれば発表時点での最高性能となる．整数性能は3.06GHzのPentium4と同程度であるが，浮動小数点性能はまさに驚異的である．それまで，トップだったItanium2(McKinley)の性能を6割以上も向上したことになる．Intelが特に強調しているのはデータベース性能(TPC-C)の高さである．TpmC*の値は21065とし，典型的なRISC(Best Published RISC：UltraSPARCか？)の2倍以上であることを示している．そのほかにも多くのベンチマーク性能を公表しており，認知度の高いベンチマークテストで高評価を受ける各種プロセッサに十分対抗できると強気である．**図23(b)**にItaniumシリーズでのSPEC CPU2000の比較を示す．

最初は低性能でどうなるかと思われたItaniumも，Merced，McKinleyを経て3代目のMadisonでかなり良い性能になった．Intelは，今回10社以上のサーバメーカの採用を受け，2003年末までにはItanium2を搭載したシステムが50種類以上出荷される見通しであることを発表した．

● 余談

2002年10月30日，米連邦地方裁判所はIntelに対しItaniumの製造，販売を差し止める判決を下した．しかし同時に差し止め命令の執行は30日間猶予され，Intelには上訴の選択肢が与えられた．

表3
従来のハードウェアをソフトウェアとハードウェアの組で実現

従来のx86ハードウェア	Crusoe	
	VLIWハードウェア	コードモーフィングソフトウェア
可変長命令のデコード	単純なデコード	x86のデコード
スーパースカラでの組み合わせ		命令の組み合わせ
スーパースカラでの命令発行		命令スケジューリング
バイパス回路		バイパススケジューリング
レジスタリネーム		レジスタリネーム
複雑なアドレシングモード		アドレスモード合成
アウトオブオーダ実行	インオーダ実行	
投機実行		投機実行
演算機能	演算機能	
レジスタファイル	レジスタファイル	
マイクロコードROM		ソフトウェアライブラリ
キャッシュ	キャッシュ	
FPスタック回路		FPスタック
		命令コード最適化

Intelを訴えたのはIntergraphで，同社は1997年頃から Intelと特許合戦を繰り広げている．2002年の4月にはPentiumシリーズに関係する特許をIntelに認めさせ，3億ドルを支払わせている．

今回の特許訴訟はItaniumがIntergraphの持つPIC(Parallel Instruction Computing)の特許に抵触するというものだ．具体的には，(1)コンパイラが並列に実行可能な命令グループを見つけて，命令コード中に付加情報として埋め込む技術(Itaniumの命令バンドル中のtemplateに対応)，(2)VLIWバンドル中の命令を機能ユニットまで届ける連想クロスバー(ItaniumのDispersal Networkに対応)が問題らしい．

連邦地方裁判所は先に，Intelによる侵害が訴えられていたIntergraph所有の2件の特許が有効であることを認定していたが，今回の判決の中でこの点を再度確認し，具体的に九つの侵害の実例を指摘した．さらに，判決とともに，差し止め命令の執行は11月29日まで猶予するとした．Intelは，特許権侵害を認めたこの判決を不服として上訴する意向を明らかにしている．結局，2004年2月11日，Intelの全面勝訴との控訴審判決が下った．

また2004年1月26日付けのCNETでは，HPがAMDのOpteronを搭載したサーバを投入する計画があると報じた．Itanium2推進の第一人者だったHPがAMDのプロセッサを採用する意味は大きい．これによりOpteronは，IBM，SUN，HPと，サーバメーカ大手4社のうち3社と契約を取り付ける形になる(残りはIntel)．

4 VLIWの実際(2)—Crusoe

● 開発背景

Transmetaは1995年にDavid R. Ditzelらが創立した会社である．Linuxの作者として有名なLinus B. Torvaldsを雇っていたことで有名である．永らく秘密裏に開発を進めてきたが，1999年の半ば頃から，L. Torvaldsが開発にかかわるMPUが密かに開発中であるという噂が流れていた(実際にはL.TorvaldsはCrusoeの開発には無関係)．そのMPUは今までになく画期的なものという触れ込みだったので業界の注目を集めていた．

そして，2000年1月，ついにTransmetaからx86互換MPUであるCrusoeの発表があった．従来，IntelやAMDは，x86アーキテクチャの命令を高速に実行するために，x86の命令をRISCライクな命令に変換し，それをスーパースカラで並列実行するという方式を採用していた．この方式はパイプラインの複雑な制御が必要で，回路規模がかなり大きくなっている．当然ながら，消費電力も大きくなる．現在のIntelやAMD製のモバイル向けMPUは，非常に優秀な省電力技術と電力消費を抑える優れた製造技術で作られているが，それでもモバイル向けとするには厳しいのが実情である．それに対し，TransmetaはVLIWを採用し，回路構成を単純にする試みを行ったのだ．

表3に示すように，Crusoeでは従来はすべてハードウェアで行っていた処理をハードウェアとソフトウェアの組み合わせで実現する．資料によれば，0.22

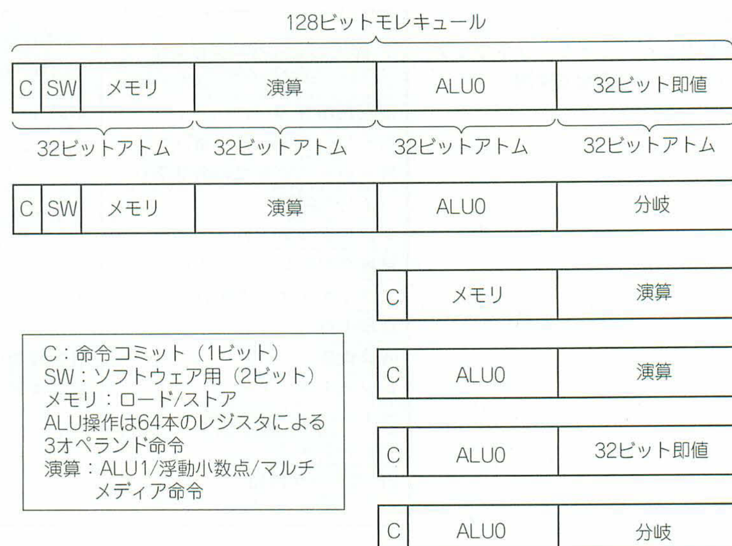


図24　Crusoeの命令フォーマット

μmプロセスで製造されるTM3120のチップ面積が77mm^2，0.18 μmプロセスで製造されるTM5400が73 mm^{2}である．0.18 μmプロセスで製造されるPentium Ⅲのチップ面積が106mm^2，Pentium4のチップ面積が217mm^{2}であることを考えると，その回路規模の小ささが実感できる．

なお，VLIWとしては128ビット長の命令を採用し，四つの操作を1命令に納める．VLIWの命令をx86命令に見せかける手法は，コードモーフィングソフトウェア(Code Morphing Software：CMS)と呼ばれるソフトウェアで，x86命令をVLIWのネイティブ命令へコンパイルするというものである.変換された命令は，メモリ上に置かれた命令キャッシュ(まともな性能を得るためには16Mバイト程度の容量が必要)に格納され，高速に実行される．Crusoeの方式をTransmetaの言葉で表現すれば「プロセッサコアに組み込まれていたマイクロコードが外部ソフトウェアとして実装された」ということになる．つまり，マイクロ命令(VLIWに変換されたx86命令)を直接実行するマイクロプログラム制御方式のMPUという考えである．

気になるCrusoeの性能だが，Transmetaの発表によると，700MHz動作のTM5400が500MHz動作のPentium Ⅲ相当ということで，それほど高性能ではない．実際の製品のベンチマークでも，600MHz動作のTM5600の整数性能が500MHzのPentium Ⅲ相当であることが実証されている．浮動小数点演算の性能は思ったよりも悪いようである．それでも，従来のモバイル向けx86互換チップと比べると，「LongRun」と呼ばれる電源管理機能によって1W以下という超低消費電力を実現しているため，ノートPCでの採用が相次いでいる．冷却用のファンが不要なことも採用の一因であろう．

なお，世の中の性能向上のトレンドはマルチプロセッサやマルチスレッドであるが，このCrusoeはマルチスレッドに対しては懐疑的である．マルチスレッドは，パイプラインがスカスカであることを補う技術であるが，VLIWで高い並列性を上げているCrusoeでは，パイプラインがフルに稼働しており，そのような姑息な技術は不要であるとしている．

● 命令フォーマット

CrusoeのVLIWエンジンは二つの整数ユニット，一つの浮動小数点ユニット，一つのメモリ(ロード/ストア)ユニット，一つの分岐ユニットから構成されている．CrusoeのVLIW命令はモレキュール(分子)と呼ばれ，最大4個のアトム(原子)と呼ばれるRISCに似た命令を含む64ビットまたは128ビット長の命令である(**図24**)．モレキュール内のアトムは並列に実行され，モレキュールの形式はアトムがどの機能ユニットに直接結び付くかを示している．これにより，デコードとディスパッチのためのハードウェアを簡略化している．

図25は，128ビットのモレキュールがアトムのスロットから機能ユニットに直接対応することを示している一例である.モレキュールはインオーダで実行され，アウトオブオーダ実行のための複雑なハードウェアは存在しない．

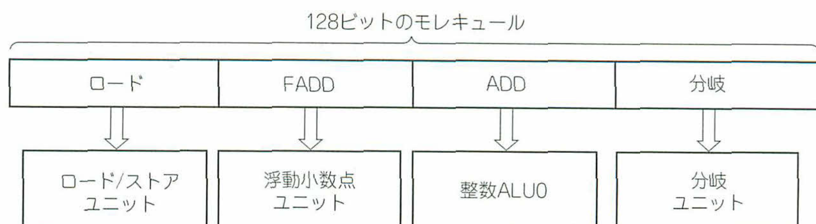


図25　モレキュール内のアトムと演算器の対応

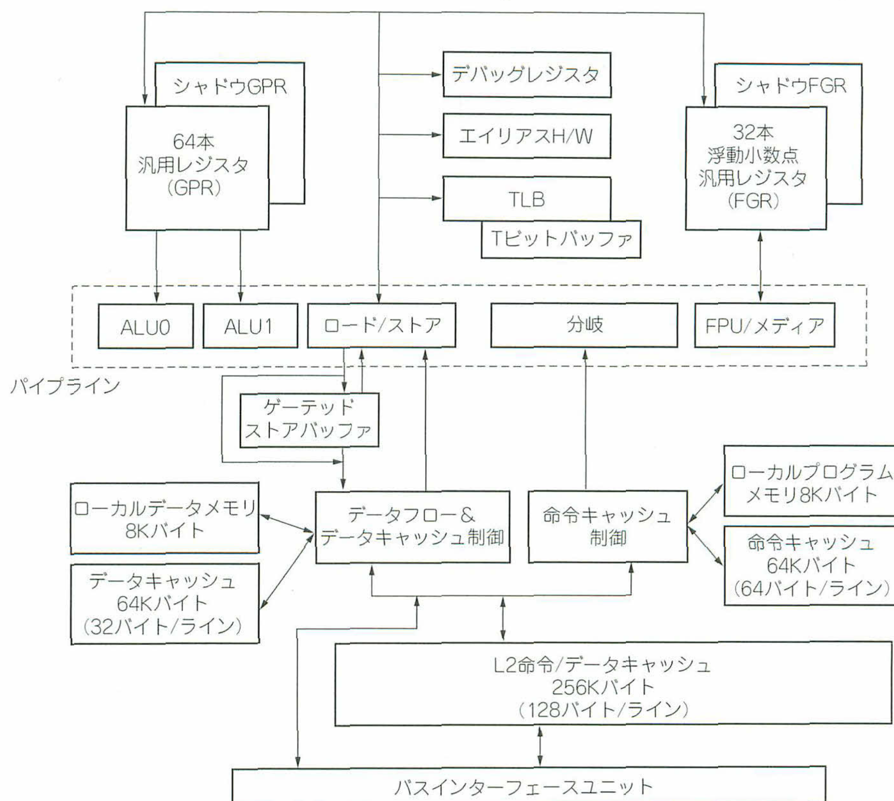


図26
CrusoeのCPUコア部のブロック図

● 内部構造

図26にCrusoeのCPUコア部のブロック図を示す．五つの並行実行可能な機能ユニットと内蔵キャッシュからなる単純な構造をしている．他のMPUがハードウェアで行っている投機実行に対応する命令列をソフトウェアで生成するため，汎用レジスタとは別に，投機実行時に参照するシャドウレジスタを，整数用48本，浮動小数点用16本内蔵する．図示されていないが，Crusoeは1チップにLongRun電力管理ユニットとノースブリッジ(PC133 SDRAM，DDR-SDRAM，PCIバス)を集積している．

● パイプライン

基本的に，Crusoeのパイプラインは，Fetch0，Fetch1，Decode，Register Read，Execute，Write backの6ステージからなるインオーダ構造をしている．一般的なx86プロセッサよりも少ないステージ数(レイテンシ)が特徴である．パイプラインは図27に示すように，各機能ユニットで若干異なる．パイプラインの最後に投機実行の終わりを明示するコミットステージが付加される場合もある．

● コードモーフィングソフトウェアの動作

コードモーフィングソフトウェア(CMS)は，x86命令からVLIW命令(モレキュール)へのコード変換用のソフトウェアである．これは，Crusoeが直接実行可能なVLIWコードで記述されている．最初，CMSはROM(1Mバイト程度の容量)に格納されており，MPUのブート時にDRAMに展開されてから実行される．CMSが格納されるメモリ領域は，同じメモリに

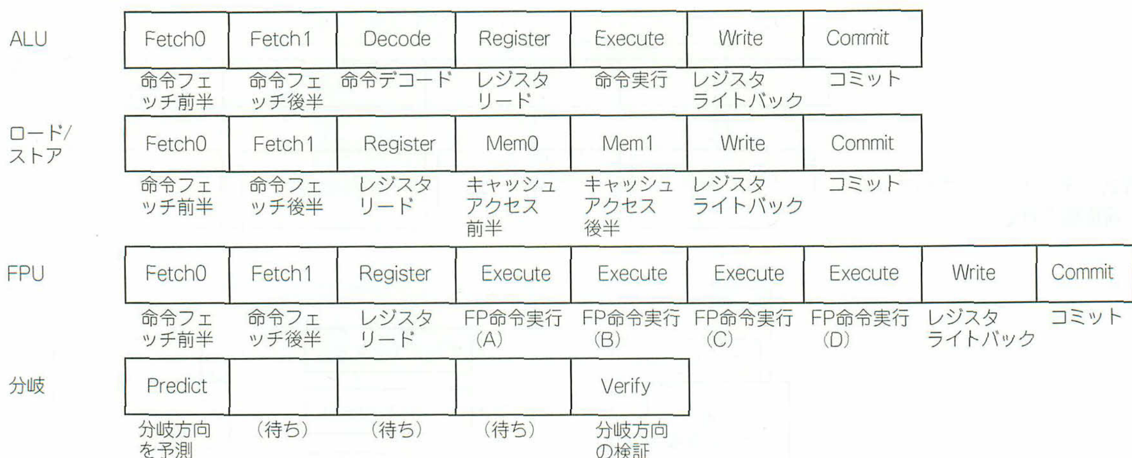


図27　Crusoeのパイプライン

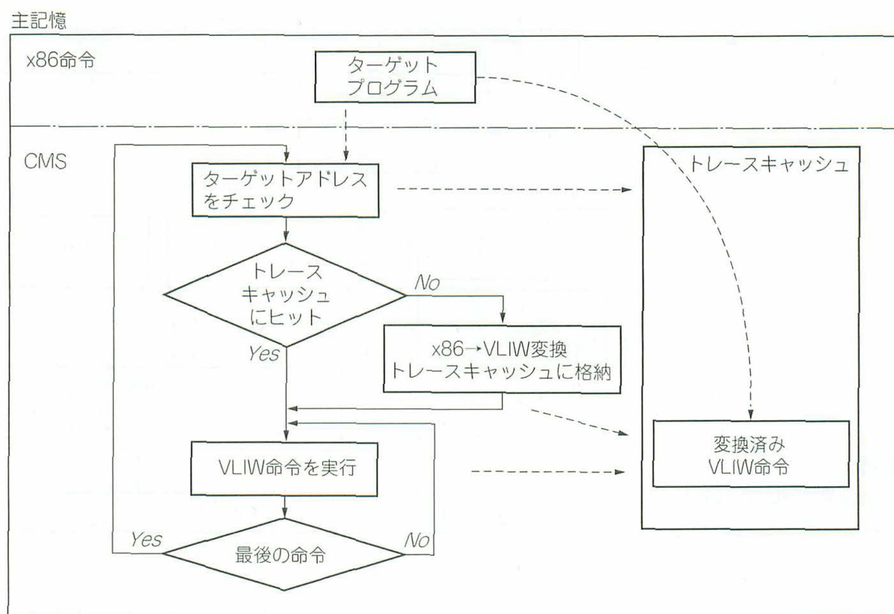


図28　コードモーフィングの動作概念

格納されているx86命令からは参照できない．

Crusoeからは，x86命令で記述されたOSやBIOS，アプリケーションプログラムはすべて，（変換して実行すべき）自分自身のアプリケーションプログラムで見える．そして，決められたターゲットアドレス（たとえば，x86のブートベクタである0xFFFF0番地）からコード変換を開始する．このとき，同じアドレスのx86命令を重複して変換しないように，変換後のモレキュール列を保存しておくための大容量（16Mバイト程度）のメモリ領域を使用する．これをトレースキャッシュと呼ぶ．

図28にCMSの動作概念図を示す．CMSは，まず変換すべきx86命令の格納されたターゲットアドレスをトレースキャッシュの中に探す．もし，ターゲットアドレスがトレースキャッシュにヒットすれば，対応するモレキュール列にジャンプしてそこを実行する．もし，ターゲットアドレスがミスすれば，CMSはx86の命令列に対し，デコード，アトムへの変換，最適化，スケジューリングを行い，新たなモレキュール列としてトレースキャッシュに格納し，そこへジャンプする．

トレースキャッシュ内のモレキュール列の実行中にコード変換した最後の命令（CMSの変換処理の先頭，またはトレースキャッシュ内の他の領域へのジャンプ命令）に突き当たると，最初の処理に戻って，次の

ターゲットアドレスがトレースキャッシュ内にあるかどうかチェックする(直接，トレースキャッシュのほかの領域へジャンプする場合もある)．

● コードモーフィングソフトウェアのコード変換例

ここでは，CMSがx86コードを対応するCrusoeのVLIWコードに変換する例を示す．次のx86命令を考える．

```
// スタックからデータをロードし，%eax に加
   える
A. addl %eax, (%esp)
// 同様に，%ebx に加える
B. addl %ebx, (%esp)
// メモリの値を %esi にロードする
C. movl %esi, (%ebp)
// %ecx から5を引く
D. subl %ecx, 5
```

最初は，変換システムの前処理として，x86の命令をデコードし，アトムの並びに変換する．レジスタ%r30と%r31がメモリロード操作のテンポラリレジスタとして使用され，次のように変換される．

```
// スタックからテンポラリにロード
ld     %r30, [%esp]
// %eax に加算し，条件コードをセット
add.c %eax, %eax, %r30
ld     %r31, [%esp]
add.c %ebx, %ebx, %r31
ld     %esi, [%ebp]
sub.c %ecx, %ecx, 5
```

次は，コンパイラでも良く知られている共通部分式の削除，分岐の削除，未実行コードの削除などの最適化が行われる．ハードウェアのみからなるx86の実行では不可能だが，ソフトウェアに基づいた変換システムなので，命令列からアトムを並び替えるだけではなく不要なアトムを削除することができる．

この例では，最後のアトム以外では条件コードをセットする必要がなく，命令スケジューリングの柔軟性が増す．また，ロードアトムの一つは冗長である．

結局，アトム列は，次のように，少ない命令に落ち着く．

```
// スタックからのロードは1回のみ
ld     %r30, [%esp]
add    %eax, %eax, %r30
// ロードしたデータを再利用
add    %ebx, %ebx, %r30
ld     %esi, [%ebp]
// ここの条件コードのみが有効
sub.c %ecx, %ecx, 5
```

最後は，スケジューラが残ったアトムを並び替え，レジスタ依存性がないアトムからなるモレキュールとしてグループ化する．この処理は，アウトオブオーダなプロセッサのディスパッチ回路で行われているのと似ている．結局，最初の命令は次の二つのモレキュールに変換できる．

```
1. ld %r30, [%esp]; sub.c %ecx, %ecx, 5
2. ld %esi, [%ebp]; add %eax, %eax,
                %r30; add %ebx, %ebx, %r30
```

このモレキュールはインオーダで実行されるが，元のx86コードをアウトオブオーダで実行するのと同じ効果があることに注目してほしい．また，モレキュール自体は明示的な並列性が指定されているので，単純なVLIWエンジンで実行できる．それゆえ，高速で低消費電力なのである．ハードウェアに命令を並び替えさせる複雑な機構は不要である．というのがTransmetaの主張である．

なお，Efficeonで採用された新しいCMSでは，最大100個のx86命令からなる領域を一度に処理し，命令処理の流れを考えた最適化を行う．また，最終的には，領域間にわたった最適化も行う．これによって著しい性能向上が果たされるという．これらの処理が4段階に分かれて行われるため，Transmataではこれを「4段ギアシステム」と呼んでいる．それなら従来方式は「3段ギアシステム」といえよう．

● 性能比較

Crusoeの性能を2001年当時のWebサイトに掲載されたベンチマークで見てみよう．HDBENCH ver3.30(整数と浮動小数点のみに着目)で比較したのが**図29**である．周波数比を考慮すれば，まあまあ健闘しているほうか．ただ，Pentium4やAthlonの動作周波数が2GHzに達しようとしている時代(2001年)において，性能的な魅力はない．

なお，Pentium MとEfficeonのHDBENCHの結果も追加しておく．HDBENCHでは，EfficeonがPentium Mに圧勝か．ループ処理に強いCMSの特性が整数性能を飛躍的に向上させているのがわかる．浮動小数点性能はやや期待外れである．

● 近年のTransmeta

時代は今より少し遡った2001年の9月．WORLD PC EXPOのために来日したTransmeta CTO(最高技

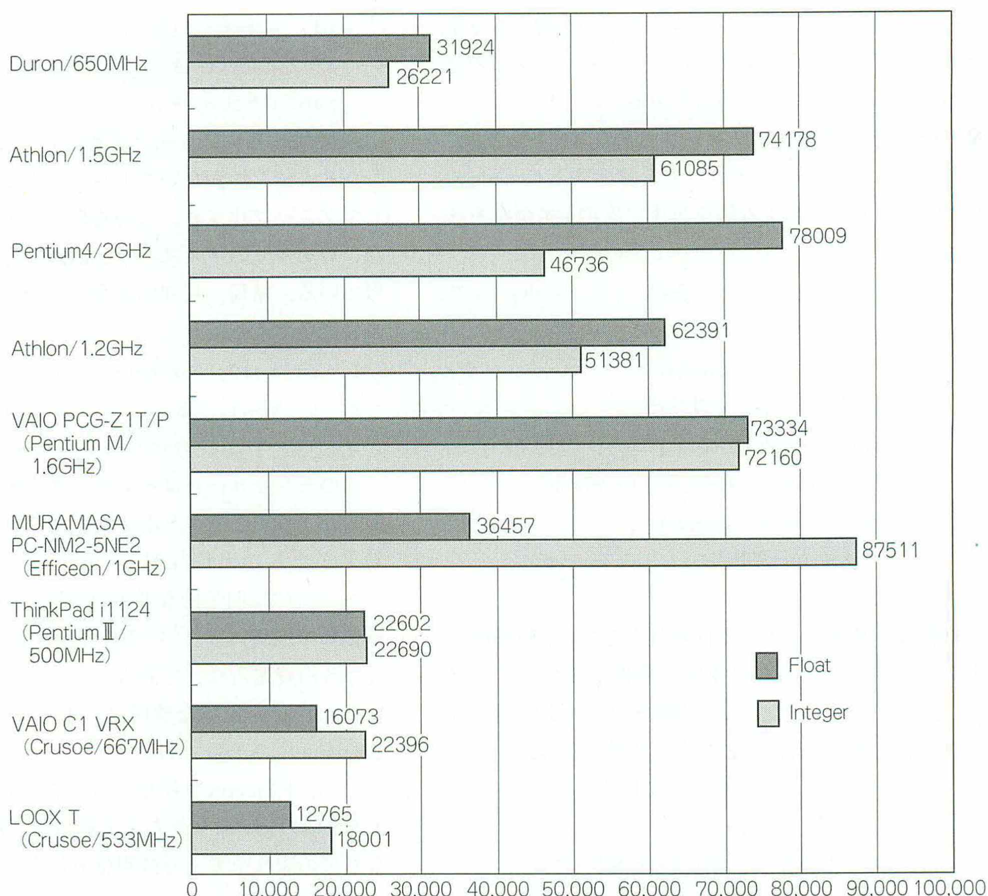


図29 Crusoeと他プロセッサの性能比較

術責任者)のDavid R. Ditzel(当時)によると，Crusoeは256ビット版のほかにも別のロードマップがあるそうだ．これは現行のCrusoeと同じ128ビットアーキテクチャで性能よりも低消費電力を追求する廉価版である．

2001年のMicroprocessor Forumで発表されたTM6000が廉価版の第1弾と見られる．従来のTM5000シリーズはNorth Bridgeまでの内蔵だったが，TM6000ではSouth Bridge，2D Graphics，PCIバス，USBなども内蔵する．DRAMをつなぐだけでPCを構成できるというのが売りである．

2002年1月の時点でFrank Priscaro Transmetaのプランディングディレクタによると，TM6000の出荷は2002年10月になり，256ビット版CrusoeであるTM8000(Efficeon)は2003年にずれ込むそうだ．また，TM6000は，従来Crusoeがターゲットとしてきた小型ノートPC以外へもプロモーションする方針らしい．周辺を内蔵するため，同等のTM5800システムより，実装面積で1/3以下，消費電力は2/3になるという．

2002年4月に開催のWinHEC(Windows Hardware Engineering Conference)では，統合型MPUのTM6000までが2003年に延期されることが明らかになった．「2002年はTM5800のクロックアップやCMSの改良にフォーカスする」というのがその公式の理由だ．

2002年5月には，CTOのDavid R. Ditzelや新CEO(最高経営責任者)のMatthew R. Perryらが東京・渋谷で記者会見を開き，苦境脱出に向けて「ジャパン・ファースト戦略」と「Crusoe1000構想」を宣言した．この「ジャパン・ファースト戦略」とは，ポータブルPCが普及している日本市場での展開を強化することで，Crusoeを全世界にも普及させていくという戦略である．そして「Crusoe1000構想」とは，価格が1000ドル以下，重さが1000g以下，電池寿命が1000分(約16時間)以上，動作周波数が1000MHz以上のPCを目指すという構想である．内容的に新規なものはないが，256ビットCrusoeであるTM8000について概要が少し公開された．TM5800と比べて，性能を2倍～3.5倍に

Column 2　Crusoeに関する個人的感想

Crusoeは，2命令または4命令の並列実行を想定しているので，Itaniumの6命令よりは現実的だと思われる．平均的には3命令同時実行のアウトオブオーダ処理と同程度の性能(同一クロックのPentium Ⅲ)と思われるが，CMSの動的な変換によるオーバヘッドのために，その75%程度の性能に落ち着くと考えられる．これは，まずまずの性能といえるかもしれない．エミュレーションでこのレベルの性能を達成できたのは快挙といってもよい．CMSがよほど優秀なのだろうか．そしてそれに付随するVLIWエンジンの処理性能の良さも忘れてはいけない．

ただ，VLIWの命令形式がハードウェア資源によって規定されるため，バイナリレベルのモレキュールの互換性を保つためにハードウェア構成を変更できないという，VLIWマシンに共通する欠点を内在しているのは確かである．その意味で，ハードウェアによる性能改善は，動作周波数の向上とキャッシュの大容量化くらいしか道が残されていない．しかし，Crusoeの位置付けはx86命令を実行するモバイル用MPUであり，VLIW命令でプログラムを書く人はまずいないと思われる．ハードウェアの変更とともにCMSも入れ替えてしまえば，バイナリ互換性の問題は解決する．もしかしたら，Torvaldsがいうように，本当に画期的なMPUなのかもしれない．

2000年の終わりに，IBMやCompaqがノートPCへの採用をキャンセルしてケチがついた格好のCrusoeであるが，その第一の理由は性能が「期待外れ」ということである．この理由はHPのItaniumの不採用にも通じるところがあるように思える．VLIWでは性能が出ないというのは定説なのだろうか．

しかし，Crusoeの利点は性能よりも低消費電力であり，日本のノートPCメーカはこぞってCrusoeの採用を表明した．米国よりも日本で人気があるのがCrusoeの特徴である．しかし，2001年になって，IntelやAMDが本気で低消費電力のMPU開発に取り組み始めると，その存在意義はなくなってきた．おりからの不況の影響で，日本でのノートPCの売れ行きは不振になり，2001年度のTransmetaの売り上げは散々なものになった．

それはともかく，Transmetaは2002年にリリースするCrusoe2.0(現在はEfficeon)で1クロックサイクル当たりの性能を倍に引き上げ，消費電力を約半分に削減するという構想を立てた．具体的には，現在では4命令128ビットのVLIWを倍の8命令256ビットに変更するという．これにより，IPCが従来の2.2から5.5に向上するという．動作周波数は1GHzを越え，性能が向上したことにより動作時間が短縮され，消費電力は0.5W以下となる予定である(これはTM8000として発表された)．また，同時にCMSも徐々に改版を行い，性能を40%向上させるという．

ここでの問題は，通常のプログラムにおいて同時実行可能な8命令の組を見い出せるか否かにかかっている．これはほとんど不可能なのではないかと思われる．個人的にはIntelの執拗な反撃にあったTransmetaが死に物狂いでロードマップを描き直しているという気がしないでもない．

VLIWで命令の並列度を上げていくという方針をIntelはMontecitoではあきらめた．しかし，Transmetaはどうするのだろう．仮にデュアルコアの場合を考えよう．この場合はCMSのトレースキャッシュ領域が2倍になって主記憶領域を圧迫してしまう．それならばVLIW単位でのスーパースカラだが，これではVLIW構成にすることで回路を簡略化し，消費電力を下げたことの意味がなくてってしまう．ItaniumとCrusoeの最大の相違点は，サーバ向けのItaniumが電力を問題としないのに対して，PCやモバイル分野をねらうCrusoeは低消費電力が売りの一つである点である．性能と電力のバランスをどのように保っていくのかがCrusoeの課題である．

また，対抗するIntelもCrusoeの躍進を快くは思っていないらしく，ノートPCに特化したBanias(Pentium M)を投入して，Transmetaの息の根を止めようとしている．2003年3月12日，日本のノートPCメーカはBaniasを搭載する機種を一斉に発表した．実機でのベンチマーク結果では，1.3GHz動作のBaniasは867MHz動作のCrusoe(TM5800)の約4倍の性能を示し，肝心な電池寿命も1.5倍以上を示した．それと前後して，TransmetaはTM8000の概要をリークし牽制を試みたが，実物がないせいか，あまり注目されなかった．今後Crusoeが，IntelのBaniasにどのように対抗していくかに要注目だった．結果はIntelの圧勝であり，Crusoe採用のノートPCは消滅した．そのノートPC市場を奪回するためにTransmetaが世に送り出したのが，かつてCrusoe2.0と呼ばれていたEfficeonなのである．

向上させるほか，消費電力は46%～67%に低減させる．動作周波数については明らかにされてないが，TM6000が1GHzであることを考えると，同程度と思われる．

2002年7月18日，Transmetaは業績不振を理由に200人(従業員の40%)の人員削減を発表した．また，家電機器向けの統合型プロセッサであるTM6000の製品化を断念し，以後はTM5800に注力することを明ら

かにした．ほかのx86互換プロセッサと比べて性能面で苦しいCrusoeの強みは冷却ファンを必要としない実装面積の小ささだったはずだが，その特徴をさらに推し進めることのできるTM6000をあきらめることで生じる波紋は非常に大きい．ほんの2か月前に発表された「Crusoe1000構想」はいったい何だったのだろう．TM6000をキャンセルした理由に対するDavid R. Ditzelの公式見解は，TM5000シリーズを使っている顧客は同じフットプリント(実装面積)のチップに同じCMS(Code Morphing Software)を継続することを望んだためであると，あくまでも顧客側の要望であることを強調している．

Transmetaは2002年11月，COMDEX/Fall 2002の会場近くのホテルで，新プロセッサAstro(後のEfficeon)を披露した．Astroは0.13 μmプロセスで製造され，「Crusoe TM8000」シリーズとして2003年半ばのリリースを予定している．Astroは256ビット版のCrusoeの第1弾である．つまり，1クロックサイクル当たり八つの命令を発行できる．Transmetaによると，Astroでは，1クロックサイクルで処理できる作業量が増加するので，消費電力の削減にもつながるとしている．これはIntelのBaniasの考え方にも通じるものがある．

さらに，Astroのデモも行ったという．デモはアプリケーションの起動を中心としたものだったが，AstroはPentium4の1.8GHzと比べて2倍以上の体感性能を得ている．それぞれのPCの動作環境(メモリ容量や動作周波数など)が明らかでないので，比較の公平性は不明だが，Crusoeとしては面目躍如である．ここで使われているCMSは，トレースキャッシュをハードディスク上にも確保して，性能向上を図っているのではないかと推測される．

2003年1月6日，Transmetaは小売店のPOS(point-of-sale)端末や業務用機器などへの採用を意図した，組み込み用の新省電力チップ「Crusoe Special Embedded(SE)」を発表した．動作周波数は667MHz/800MHz/933MHzで，それぞれレギュラーモデルと省電力モデルが用意されるという．CrusoeSEは，組み込み市場で主流のARM/MIPS製MPUと，Intel/AMD製MPUの間のニッチ市場をねらった製品であると明言している．つまり，ARM/MIPSより性能(周波数)が高く，Intel/AMDより電力が低いことが売りである．Transmetaのニュースリリースを読んでみても，CrusoeSEが従来品とどう違うのかあまり判然としない．組み込み向けに温度拡張(最大100℃)がされている程度であろうか．それにより，動作周波数が従来品より低くなっている．裏をみるなら，PC分野ではIntelやAMDに敵わないと予想したTransmetaが，主要な応用分野を組み込み制御分野に方針転換しただけだと取れなくもない．

2003年の1月14日には，従来品のTM5800に，暗号化エンジンやディジタル認証と暗号化キーを保存できる機能などの複数のセキュリティ機能を組み込むことを表明した．具体的には，DES(Data Encryption Standard)やDES-X，Triple-DESのアクセラレータと，保護されたメモリ領域を内蔵する．これらは，主に無線ネットワーク上で使われるノートPCのデータ保護のための利用を意図しているらしい．セキュリティ機能を別チップや別ソフトで実現しているIntelと差別化する目的もある．この新プロセッサは，すでにサンプルが完成し，2003年4月から6月にかけてPCメーカに大量出荷が予定されているという．Transmetaもまだ PC分野を諦めていないという決意表明か．

● Efficeonの発表

2003年8月12日，TransmetaはTM8000のブランド名を「Efficeon」とすることを発表した．Efficient Computingの意味をもつという．Efficient(効率的)とはエネルギーの効率的な利用を示す．Transmetaはコードネームである Astroのもじり(Astrino，Aztro，…)を検討していたが，商標に抵触するものばかりで断念したという．

そのEfficeonは動作クロックが1GHz以上の性能で登場し，HyperTransportに対応すると予想されている．8月12日の発表では，Efficeonは従来のCrusoeに対して同一動作周波数では実用アプリケーションで50％，マルチメディアアプリケーションで80％の性能向上が期待できるとしている．

かくして，2003年10月14日，Microprocessor ForumでEfficeonの詳細が発表された．それによると，Efficeonは2003年内に1.3GHz動作で出荷されるという．NorthBridgeを内蔵することは従来のCrusoeと同じだが，HyperTransportとAccelerated Graphics Port(AGP)を内蔵して性能向上を図っている．Transmetaによれば，従来のCrusoeはAGPをサポートしておらず，それがノートPC市場進出への足枷になっていたという．

Efficeonの特徴は，従来のCrusoeが4命令(アトム)を並列実行していたのに対し，8命令を並列実行するものである．しかも，演算器は11種(ロード/ストア

/32ビット加算器×2，整数ALU×2，エイリアス×1，制御×1，浮動小数点/MMX/SSE/SSE2×1，MMX/SSE/SSE2×1，分岐×1，実行×2)に増加され，8個並んだ任意のアトムから任意の演算器に命令発行可能だという．従来はアトムの位置によって選択される演算器が一意に決まっていた．構造的には，一歩Itaniumに近づいた感がある．

なお，「エイリアス」はアドレスの一致性をチェックする演算，「制御」は内部レジスタの制御，「実行」はインタプリタ(Efficeon)のループ処理の効率的な実行を支援するものである．

Efficeonではデータの供給を早めるために，ストアの次にロードが来た場合は，ロードを優先して実行する．しかし，ストアとロードのアドレスが一致する場合は，このような追い越しができない．このアドレスの一致判定をするのが「エイリアス」ユニットである．このようなロード/ストアの並び替えはI/Oポートに関する場合は矛盾を来たすのだが，メモリとI/Oの区別はCMSが「4段ギア」の「一速」目に認識して矛盾が発生しないようにしている．

EfficeonのVILW命令には「execute」という命令が追加されており，これを実行するのが「実行」ユニットである．インタプリタ(CMS)が内部ループを発見すると，その命令列を「execute」という1命令で代替させるのだという．「実行」ユニットの具体的な構造はよくわからないが，同じ命令列の繰り返しを計数するカウンタなどがあるものと思われる．

また，内蔵キャッシュもL1命令キャッシュが128Kバイト(4ウェイ)，L1データキャッシュが128Kバイト(8ウェイ)と倍増され，L2キャッシュが1Mバイトと4倍になっている．

さらに，浮動小数点演算の性能向上を目指したのか，浮動小数点レジスタが64本に倍増されている(シャドウレジスタは48本)．

CMSもバージョンが5.0になり，繰り返し処理の最適化が図られている．つまり，同じ命令列の繰り返し実行回数がある程度大きくなると，トレースキャッシュ内の命令(アトム)をスケジューリングし直して実行するのだそうだ．このへんの説明はやや理解不能だが….

パイプラインは，整数演算が6ステージ，浮動小数点演算が8ステージである．これは，Crusoeに比べてパイプライン段数が1段減ったようにみえる．実際は，Efficeonでは，命令フェッチと実行がデカップル構成になっているため，命令フェッチを行う2段分のステージが省略されているので，1段増加しているともいえる(図30)．

IS：Instruction Issue
DR：Instruction Decode
RM：Register Read for ALU operands
EM：Execute ALU opration
CM：ALU Condition flag completion
WB：Writeback Results to Integer Register File

(a) 整数パイプライン

IS：Instruction Issue
DR：Decode-1
DT：Decode-2
XA：Floating Point Compute Stage-1
XB：Floating Point Compute Stage-1
XC：Floating Point Compute Stage-1
XD：Floating Point Compute Stage-1
WB：Writeback Results to FP Register File

(b) 浮動小数点パイプライン

図30　Efficeonのパイプライン

● Efficeonの命令フォーマット

TransmetaのDitzelによると，Efficeonは，Crusoeが64ビット(二つのアトム)と128ビット(四つのアトム)の命令長だった(図24参照)が，Effiecon では，最小32ビット(一つのアトム)から最大256ビット(八つのアトム)まで，32ビット単位に8種類の命令フォーマットをサポートするそうである．これはCMSでx86命令から変換されたアトムをトレースキャッシュに格納するオブジェクト効率を高める(トレースキャッシュの容量を節約する)ためである．

Crusoeでは，かならず二つまたは四つのアトムでモレキュールを構成しなければならないため，並列度が少ない場合は，空いたスロットにNOP命令を詰めなければならなかった．Diztelによると，Crusoeではトレースキャッシュ内の20%～25%がNOP命令であった．それが，Efficeonでは約5%になり，通常のRISCプロセッサと同程度になったという．

このDitzelの説明は衝撃的である．VLIWでNOP命令の存在確率が多いということは，4命令が並列実行できる機会が少ないということを意味する．20%～25%がNOP命令ということは，せいぜい3命令の並列実行しかできていなかったということである．素直に取ればVLIWは無効という告白にもなる．なのに，Efficeonであえて8命令の並列実行フォーマットに拡張するということは，オブジェクト効率を悪化させる方向に進んだように思える．それを解消するための手段が，命令発行の単位を1～8アトムと，柔軟性を持

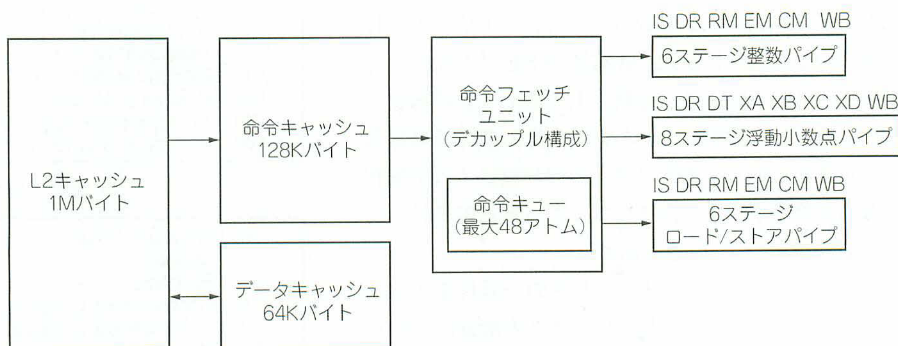


図31
Efficeonパイプラインの命令実行フロー

たせることである.

しかし，命令発行の単位が1～8アトムということは，アトムを命令の単位とする8ウェイのスーパースカラと同じではないのか．多分，VLIWというからには，各アトム間の依存性はチェックしないのだろう．そう考えれば，モレキュールを構成する最大八つのアトムのそれぞれが任意の演算器に発行可能な理由がわかるような気がする．

図31にEfficeonパイプラインでの命令実行フローを示す．デカップル方式なのは，IntelのItanium2も同じなので驚くべきことではないが，命令キューの格納単位がアトムにとされているところが，通常のスーパースカラ構造のプロセッサを連想させる．

結局は，EfficeonもVLIWをあきらめて，スーパースカラを採用したのだなと考えると一抹の寂しさを感じる．まあ，TransmetaとしてはVLIWが大事なのではなく，いかに低消費電力で高性能なプロセッサを提供するのかが目的なので，VLIWかスーパースカラかというのはそんなに大きな問題ではない．しかし，EfficeonでもVLIWと大きく宣伝している….

一方，Crusoeで強調していたVLIWで消費電力が下がるという主張はウヤムヤになり，LongRun2という荒業でスーパースカラ構造にしたことによる消費電力の増加を削減するのかと考えると，Efficeonも普通のプロセッサになってしまったなと思ってしまう．

構造的には，ライバルであるPentium Mとほとんど変わらなくなったので，それと同性能以上の性能を達成できたとしても不思議ではない．結局，VLIWではだめだったのか．

● **Efficeonのラインアップ**

最初のEfficeonはTSMCの0.13 μmプロセスで製造されていた．L2キャッシュの容量とパッケージの展開により，TM8600(L2キャッシュ1Mバイト，通常パッケージ)，TM8300(L2キャッシュ512Kバイト，通常パッケージ)，TM8620(L2キャッシュ1Mバイト，小型パッケージ)の3種類がある．消費電力(TDP)は，動作周波数が1.3GHz，1.2GHz，1.1GHz，1.0GHzに対して，それぞれ14W，12W，7W，5Wという．

その後，富士通の90nmプロセスで製造したEfficeonには，TM8800(L2キャッシュ1Mバイト，通常パッケージ)，TM8500(L2キャッシュ512Kバイト，通常パッケージ)，TM8820(L2キャッシュ1Mバイト，小型パッケージ)の3種類がある．消費電力(TDP)は，動作周波数が2.0GHz，1.8GHz，1.6GHz，1.4GHz，1.0GHzに対して，それぞれ25W，12W，7W，5W，3Wという．

Transmetaは消費電力の小ささを強調している．しかし，従来のCrusoeが7W以下だったことを考慮すると，周波数比から考えて，それほど小さい値だとは思えない．さすがに90nmプロセスになると電源電圧が低下するせいで少し小さくなっているようだ．

Transmetaは消費電力を下げるために，トランジスタのスレッショルド電圧(V_{th})をソフトウェアで制御することにより，トランジスタのゲートリーク電流を減らすことが可能なLongRun2技術を提唱している．本当にそのようなことが可能なのか(単なるバックゲートバイアスという憶測もある)と疑問を抱く．このLongRun2は130 μmプロセスのEfficeonには搭載されず，90nmあるいはその先の65nmプロセスからの実装という．しかし，そのための回路は最初のEfficeonから実装されているという．

2003年10月のMicroprocessor Forumで公表されたLongRun2技術は「全体像の10%にすぎない」という話であり，水面下にはさらに画期的な電力低下技術が隠されている可能性がある．TransmetaはLongRun2技術を他社にライセンスすることを考えており，その

ライセンス収入をEfficeonと並ぶ二本柱にしようと考えているようだ.

日本法人トランスメタの社長は，既存のCrusoeとEfficeonの棲み分けについて「Crusoeはローエンドバリューマーケットをターゲットにし，組み込みシステムを中心に薄型PCやハンドヘルドPCまでをカバーする．Efficeonはハイエンドパフォーマンスマーケットを対象とし，従来のCrusoeの市場であった組み込みシステムからウルトラライトノートPCやメインストリームノートPCに加え，静音タイプの省スペース型デスクトップやブレードサーバ，ディジタルコンシューマと呼ばれる家電まで，広範囲な市場をターゲットにする」と説明している．しかし，それはEfficeonのベンチマーク性能(後述)を見る限り，無謀な計画と思われる．

TransmetaはEfficeonを採用する大手OEMベンダーとして，シャープ，HP，富士通を挙げている．業界の噂では，シャープはMURAMASAの後継機種に，HPはCompaq Tablet PC TC 1000の後継機種に採用を検討しているらしい．富士通はまだ検討段階に入ったところだという．しかし，こういうポータブルPCとかミニノートPCという製品群は，日本以外でのシェアは1%未満といわれており，Efficeonの出荷台数は期待できない．これをどう乗り切るかがTransmetaの手腕の見せどころであろう．

Efficeonの特徴は，(ある程度の)低消費電力(冷却ファンの削除)と，NorthBridgeとAGP内蔵によるシステムコストの安さにある．HPはブレードPCという新分野にEfficeonを採用したが，1GHz動作のPentium M採用のシステムが約2000ドルなのに対して，1.8GHzのEfficeonのシステムなら約1000ドルであると強調している．

● Efficeonの性能

さて，現在のCrusoeがノートPCから消え失せていった背景には，対抗機種のPentium Mと比べて性能が劣るという点があった．Efficeonが世間に受け入れられるか否かは，Pentium Mを凌ぐ性能を達成できるかどうかにかかっている．なお，VLIWの幅が2倍になったことによる性能向上は平均的には1.5倍ということである．

Efficeonの発表と同時に，Transmetaはベンチマーク結果を公表している．1.1GHz動作のEfficeonと900MHz動作の低電圧版(ULV)Pentium Mを比較している．その理由は，どちらもTDPが7Wであり，その電力を「ファンレスノートの限界」とTransmataが考えているためだそうである．しかし，1GHz動作のULV Pentium MのTDPも7Wであり，やや公平さを欠く．

結果としては，ベンチマークによって得手不得手はあるものの，TM8600はPentium Mの約1.1倍の性能であった．IntelやAMDと比べて同一動作周波数での性能が低いと揶揄され続けてきたTransmetaにとって面目躍如というところである．しかし，Astroのリーク時点では1.8GHz動作のPentium4の2倍の体感性能といっていた事実はなかったことになってしまった．

また，TransmetaはTM8600とPentium Mはほぼ同性能であるが，アイドル時の消費電力はTM8600では1/8と強調する．つまり，Pentium Mは1.45WであるがEfficeonは0.18Wであるという．

Efficeon(TM8600/1GHz)搭載の「MURAMASA PC-MM2-5NE2」は2003年12月8日に発表された．しかし，どのWebサイトもこのMURAMASAのベンチマーク記事を載せないのが不気味である(シャープからベンチマーク禁止令でも出ているのか？)．ただ，あるWebサイトの記事によると，Crusoe(TM6500/1GHz)と比較して，OSブートでは33%，PowerPointの2回目起動では52.5%高速化されたという報告がされている．このような比較に意味があるかどうかはわからないが，堂々と高性能を謳わないところに先行きの不安を感じさせる．

そのWebサイトの2003年12月17日付けの記事には，満を持して(？)，Efficeon(シャープ MURAMASA)のベンチマーク結果が載っていた．結果としては，やはり，900MHz動作のPentium Mと同等の性能である．その記事が電池の持ち時間をやたらにアピールしているところにうさん臭さを感じるが．

● Efficeonのロードマップ

図32にEfficeonのロードマップを示す．最初の製品にはTSMCの130nmプロセスを利用するが，2004年の下半期には富士通の90nmプロセスに移行する．何故富士通なのか．それは時代は130nmから90nmに移行しようとしているが，Transmetaが各社の90nmプロセスをベンチマーキングしたところ富士通のものがもっとも性能がよかったためという．Efficeonの売りは低消費電力なので，富士通が気を利かしてスレッショルド(V_{th})の高いプロセスを提示したが，TransmetaはV_{th}を制御する技術(LongRun2)があるので，できるだけV_{th}の低い高速プロセスを望んだというの

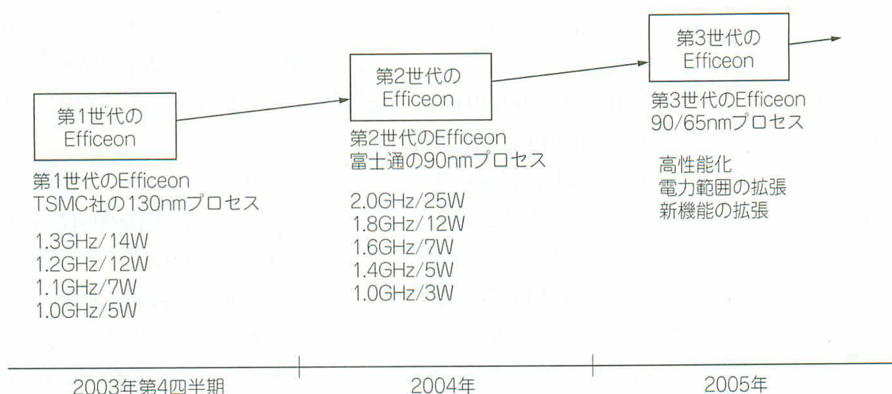


図32 Efficeonのロードマップ

は，業界では有名な話である．

Microprocessor Reportは，Transmetaが自社のロードマップどおりに製品を開発できれば，IntelのPentium Mに奪われた市場をいくらか取り戻せると報じている．130nmで製造する1.3GHz動作のEfficeonは，1.7GHz動作のPentium M(消費電力は25W)に比べると動作周波数が低い．しかし，90nmで製造する2GHz動作のEfficeonは，同じ90nmの次世代Pentium M(Dothan)に十分対抗できる．Crusoeに比べるとEfficeonは性能面で大きく飛躍した．Efficeonを搭載したシステムが登場してPentium M搭載システムと比較できれば，Transmetaの主張の正しさを検証できる．現在のところは，同社のロードマップと技術は極めて有望に見えると結んでいる．

● 余談

Crusoeの由来はいうまでもなく，ロビンソン・クルーソーである．世界中誰でも知っていて，数々の苦労を創意工夫で乗り越えて最後に目的(故郷に帰る)を達成したという点にならい，世界でもっとも一般的なMPUとなるべく命名されたという．その名称の通り，Crusoeの前途には波乱万丈が予想される．

…と思っていたら，後継機種は「効率」をもじったEfficeonである．上述したように，Efficeonは8ウェイスーパースカラともみなせるので，Crusoeはよほど効率が悪かったのだろう．

まとめ

かつて，VLIWという方式を初めて耳にしたとき，頭に浮かんだのはマイクロプログラムというイメージだった．実際，固定長の命令，同時実行できる複数の命令をひとまとめにして処理するというのはマイクロプログラムの発想そのものである．

そこで，本章ではマイクロプログラミングとVLIWを同時に取り上げて対比してみたが，いかがだったであろうか．違いは，マイクロプログラミングは人間の手で泥臭い方法で作成されるが，VLIWはコンパイラを使うというところであろうか．性能を出すために人間の頭脳とソフトウェア(コンパイラ)が対峙しているイメージには興味深いものがある．ただ，現在脚光を浴びているItaniumやCrusoeがこのまま生き延びていけるという意見には，個人的にはやや懐疑的なことを付け加えておく．最終的には絶対的な性能がものをいうような気がするので．低消費電力はあくまでオマケだと思う．

しかし，Transmetaの行き先としては低消費電力をアピールせざるをえない状況にあるのは確かである．識者の予想には，Efficeonはブレードサーバに使われることを意図しているというものもあり，Transmetaの目指す応用分野がよく見えない．

VLIWは，PCの世界での普及はまずないと思われるが，組み込み制御用のプロセッサにはVLIW構造を採用するものが数多く見られる．汎用プロセッサでは，富士通のFR-Vや三菱電機のM32Rがある．MPEG-4など専用分野に特化したものは枚挙に暇がない．やはり，単純な構造でそこそこの性能を得られる点が魅力なのだろう．

Appendix 4

誤り検出/訂正符号やシステムの多重化など

高信頼性をサポートする機能

コンピュータの応用は，さまざまな分野に広がっている．その中でも，金融機関のオンライン処理，医療機器，ロケットや人工衛星，交通機関制御への応用は高い信頼性を要求される．これらの分野では，コンピュータが停止すると重大な事故を引き起こしてしまう．しかし，どんなに注意していても故障(フォールト)は発生する．その場合でも，被害を最小限に食い止めるしくみがコンピュータに求められる．このように，故障に強く無停止動作を実現するシステムを「フォールトトレラントシステム」という．

また，大型計算機やEWSなどの一般的なコンピュータでも，ある程度の高信頼性は重要である．いったん故障が発生すれば，修復や保守のコストが高くついてしまう．それを避けるためのしくみは，RAS(信頼性：Reliability，可用性：Availability，サービス性・保守性：Serviceability)として，高性能コンピュータの特徴の一つとなっている．

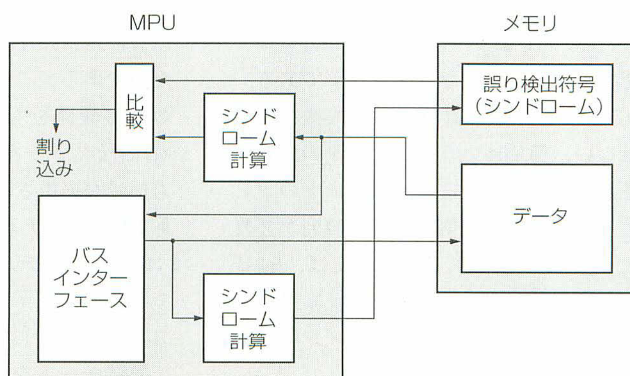
高信頼性は，MPU，メモリ，記憶装置，I/O装置など，システムのすべての構成要素に要求される．その本質は故障の検出にある．ここでは，MPUが提供するフォールトトレラントシステムのサポート機能について説明する．

● 誤り検出/訂正符号

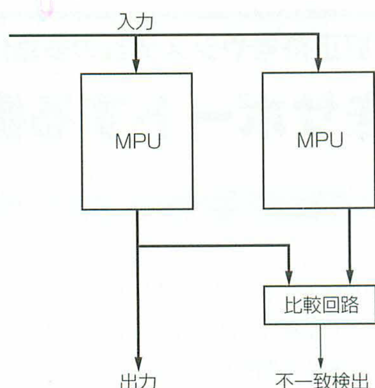
MPUに直接接続されている周辺機器には，メモリとI/Oがある．I/Oに関しては，同じアドレス(ポート)であっても入出力される値は場合によって異なるので，その値が正しいかどうかを判断する方法はない．しかしメモリに関しては，与えられたアドレスに対するメモリの内容は意図的に変更しないかぎり不変であるはずなので，その値が正しいか否かを判別するのは重要である．メモリに記憶されているデータは放射線やノイズによって破壊されることもあり，MPUがリードする値がいつも正しいとはかぎらない．高信頼化システムではメモリ内容の正当性を保証する必要がある．

メモリの誤りを検出する方法として通常は以下のような作業が行われる．まずメモリライト時に，そのデータを加工した特殊な値(シンドロームと呼ばれる)をデータと一緒にメモリに格納しておく．その後のメモリリード時に，メモリからデータと同時にリードしたシンドロームとデータから新たに計算されるシンドロームを比較し，一致するか否かを検査する(**図A**)．シンドロームが一致すればそのデータは正しいとみなせる．この際，データからシンドロームを再計算する機構と比較する機構はMPU内部に備わっている．誤りを検出した場合は例外を発生する．

このような誤り検出符号では，パリティとECC(Error Checking and Correcting)が有名である．ただ，パリティにしろECCにしろ，シンドロームを計算する機能が高速動作時においてクリティカルパスとなるため，組み込み用途などの安価なMPUでは採用されない．



図A
誤り検出符号を使用するシステム



図B　冗長構成(外部回路で不一致検出)

▶ **もっとも単純なパリティ**

誤り検出符号でもっとも単純なものはパリティである．これは，データ内の全ビットの排他的論理和を取った1ビットの値である．パリティを含めて結果が0となるもの(つまり1の数が偶数)を偶数パリティ(even parity)，結果が1となるもの(つまり1の数が奇数)を奇数パリティ(odd parity)と呼ぶ．

パリティは，その生成原理から，偶数個のビットが誤った場合でも正しいデータとみなしてしまう．その危険を少しでも低減するため，データをいくつかに分割して，それぞれをパリティで管理する．たとえば，32ビットデータであれば4分割して，8ビットずつにし，対する計4ビットパリティを用いる．

▶ **パリティより複雑なECC**

ECCはパリティよりも複雑な符号化を用いる．パリティと異なり，データに誤りがあった場合それを訂正することができるのが特徴である．何ビットの誤りを訂正できるかにより，いろいろな符号化方法があるが，実現のしやすさとハードウェア規模を考慮してSEC-DED(Single-bit Error Correcting and Double-bit Error Detecting)コードが多用される．これは，その名のとおり，1ビットまでの誤りを訂正し，2ビットまでの誤りを検出することができるコードである．

SEC-DEDのシンドロームの計算方法は複雑なのでここでは言及しない．簡単にいうと，データの各ビットを数種類の係数列と積和することで数ビットのシンドロームを得ることができる．何種類の係数列が必要かはデータのビット長に依存する．たとえば，64ビットデータに対しては8系列が必要である．結果として，64ビットデータからは8ビットのシンドローム(ECCコード)が生成される．

メモリに保存されているECCコードと計算したECCコードの排他的論理和を取った場合，結果が0であればデータは正しいとみなせる．結果が0でない場合は，それをデコードすることでメモリデータ長に等しい値を得ることができる．もし，その値の中に1であるビットが一つだけあれば，メモリデータの対応位置のビットが誤りである．つまり，元のデータと排他的論理和を取ればデータを訂正できる．もし，1であるビットが複数あれば，メモリデータが誤りであることを表す．この場合は対応するビット位置が誤りというわけではない．

ECCコードを採用する場合，メモリリードと同時にデータを訂正するには厳しいタイミングが要求されるため，動作スピードに影響を与えることになる．そこで，MPUへの実装では，ECCコードの排他的論理和を取った時点で，その値が0でなければ例外を発生し，訂正処理をソフトウェアに任せることが多い．

● **冗長性による高信頼化**

MPUの高信頼化では，共通の入力に対し複数のMPUを同時実行させ，各MPUの出力を比較し，その一致を検査する．これをロックステップ(lock-step)操作という．**図B**のように外部回路で一致を検査する方式もあるが，回路規模が大きくなるため好まれない．通常はMPU自体が監視モードをもっている．

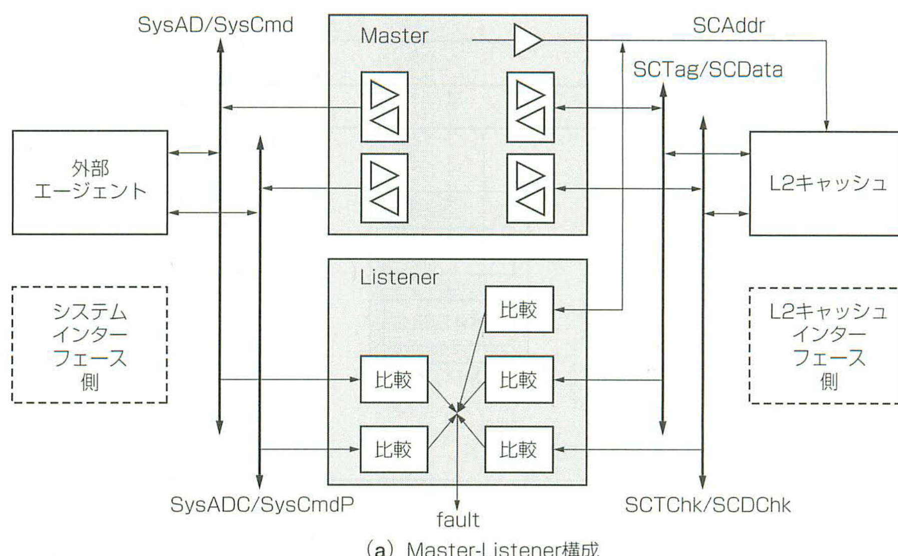
監視モードでは，MPUの出力端子は入力端子に切り替わり，対応する端子の出力(これが監視モードへの入力)と自身の出力をMPU内部で比較する．もし，不一致が発生すれば，専用端子を活性化して，故障の発生を外部に通知する．

ロックステップ構成における注意点は，複数のMPUを完全に同期化して動作させる必要があるということである．とくに割り込みなどの非同期入力に関してはタイミングがずれないように注意をしないと，MPU間の動作がずれてしまうことがある．この場合，たとえ正常動作していても不一致が生じることがあるためである．このため，MPU間で定期的に同期を合わせる機構をもつMPUもある．たとえば，特定のMPUをストールさせて，待ち合わせを行うための入力信号が提供されるのである．

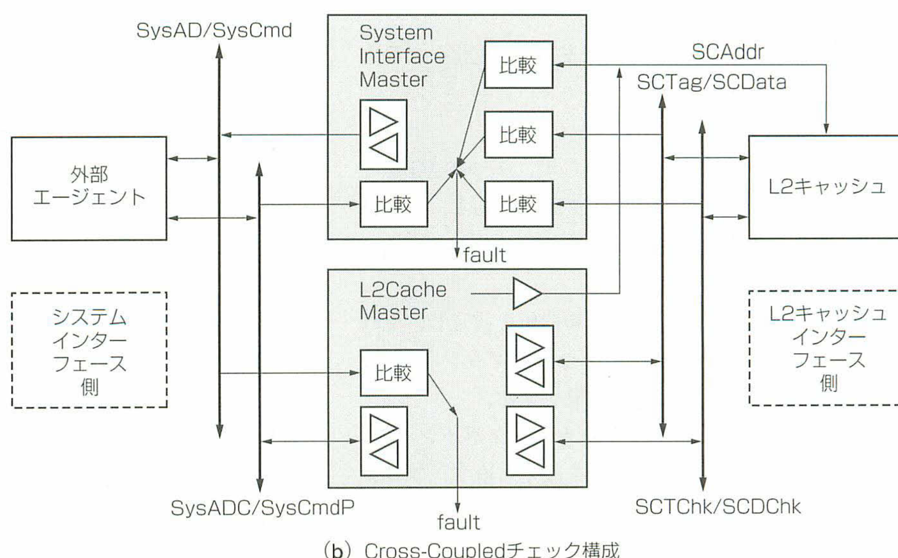
▶ **2重化システム**

2重化システムでは，二つのMPUを並列に接続してロックステップ操作をする．一方が通常モード，片方が監視モードである．

監視モードをもつMPUとしてはMIPSのR4400が



(a) Master-Listener構成



(b) Cross-Coupledチェック構成

図C
2重化システムの例

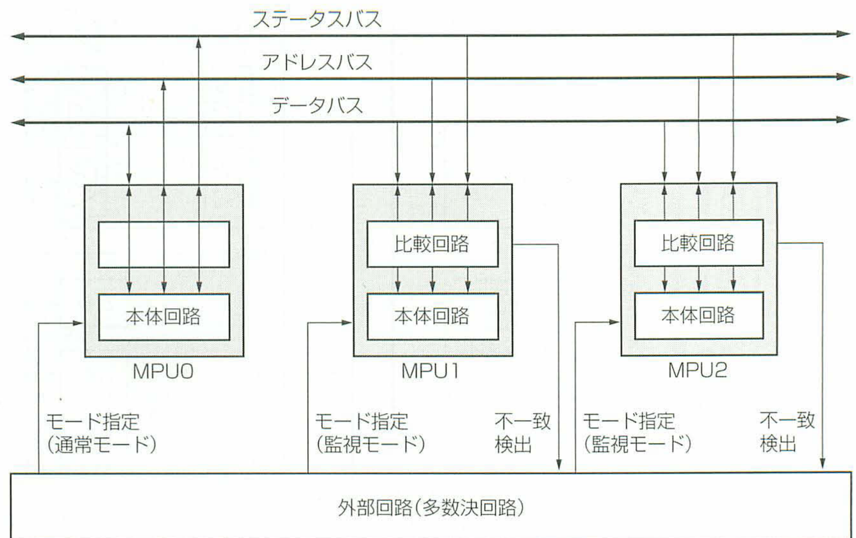
ある．R4400では2種類の2重化システムをサポートする．一方が通常動作をし，片方が監視動作をするMaster-Listener構成〔**図C(a)**〕と，一方がシステムインターフェースを駆動しながらL2キャッシュインターフェースを監視し，片方がシステムインターフェースを監視しながらL2キャッシュインターフェースを駆動するCross-Coupledチェック構成〔**図C(b)**〕である．つまり，通常モードを含めて4種類の動作モードをもつことになる．R4400はブート時に4種類の動作モードを指定できる．

▶3重化システム

2重化システムでは，通常モードと監視モードのどちらのMPUが故障したのか知ることができない．そこで，MPUを三つ並列接続して多数決で故障したMPUを特定する構成がある．一つが通常モードで，残りの二つが監視モードである．この構成は，一般にTMR(Triple Mudular Redundancy)構成(**図D**)として知られている．

この構成では，監視モードの一つのMPUが故障した場合，構成を2重化システムに変更してある程度動作を継続できるという利点がある．その間に，部品の交換や修理を行うことができ，時間稼ぎができる．

宇宙開発事業団(NASDA)が打ち上げを行ったH2Aロケットの姿勢制御，エンジン制御にはNECのV70が使用されている．V70もFRM(Function Redundancy Monitor)と呼ばれるロックステップ動作をサ



図D　3重化システムの例

ポートする．H2AロケットではV70の3重化システムが使われているそうである．

● 監視タイマ

複数のMPUで冗長構成を採らず，一つのMPUで故障検出を行う方法として監視タイマがある．これはウォッチドッグタイマ(Watchdog Timer)として知られている．

これは単純なタイマである．初期値を設定し，それがカウントアップあるいは，カウントダウンされて，一定の値に達すると割り込みを発生する．

プログラムでは，いくつかのチェックポイントで，ウォッチドッグタイマに初期値を設定し直す．プログラムの実行に何か不都合が発生し，ある時間内にウォッチドッグタイマを設定できなければタイマが規定値に達し，割り込みが発生して故障を通知するしくみである．故障発生時はMPUをリセットする必要があるので，ウォッチドッグタイマを専用にもつMPUでは割り込みの代わりにリセット例外を発生するものもある．

＊　　＊

フォールトトレラントシステムにおいて，MPUに要求される機能について説明してきた．使い捨ての感がある組み込み機器やPCでは，低コスト化の要求が強いため高い信頼性を提供することは少ないが，こういう世界もあることを知っておいてほしい．

第8章 処理性能を上げるための最後の切り札
マルチプロセッサの基礎

MPU単体での性能向上に限界が見え始め，さらなる処理性能向上のために，複数のプロセッサを使って並列実行する方法(マルチプロセッサ)が考え出された．マルチプロセッサには，いわゆるプロセッサを複数個並べたものから，1チップの中にコアを複数個実装したものなどがある．マルチプロセッサ時のメモリ共有やキャッシュについても考察している．

2001年，PCの世界ではAMDのAthlonMPやIntelのXeonが発表され，これ以降，デュアルプロセッサ構成がにわかに脚光を浴びてきた．これらは二つのプロセッサをSMP形態で構成するマルチプロセッサである．マルチプロセッサは，従来はワークステーションやサーバなどのハイエンド専用だったが，現在ではデスクトップ用の可能性も見えてきている．つまり，AMDやIntelの製品系列にはデュアルプロセッサだけではなく，4CPU以上のマルチプロセッサも標準的に組み込まれている．

MPUの高速化技術はもはや出尽くした感があり，マルチスレッディングやマルチプロセッサが最後の手段と考えられる．

本章ではマルチプロセッサ構成の基礎について説明する．マルチプロセッサ構成自体には多くの研究がなされており，最新の成果を説明することは難しい．しかし，その基本となる知識はそれほど多くはない．それらについてみていってみよう．

なお，ここではスーパーコンピュータのような超並列構造には言及しない．

1 マルチプロセッサの基礎

● マルチプロセッサ

MPUの実行速度を上げる手法としてパイプラインやスーパースカラが考案されてきたが，それらはプロセッサ自体の高速化であり，それ以上に性能を追求する場合，単一プロセッサでは限界がある．そこで，タスクをいくつかのプロセスに分割し，それぞれを別個のプロセッサで並列に実行するという方式が考えられた．これがマルチプロセッサである．

RISCの産みの親であるStanford大学のHennessy総長は次のように言っている．「個別の技術でMPUの性能を向上する手法は行き詰まった．これからは，マルチプロセッサ構成のMPUを効率的に利用するソフトウェアの開発が性能向上のキーポイントである」と．もっとも，HennessyはR10000を発表した時点(1992年)で，性能向上の次の技術は1チップに複数のプロセッサを統合することだと主張していた．諦めるのが早すぎる感もあるが，それ以降登場してきた技術はチップマルチプロセッサとマルチスレッディング(ハイパースレッディング)のみ(その意味でHennessyの予言は的中している)であり，ここ10年以上飛躍的な技術革新がないのも事実である．

● マルチプロセッサの構成方式

一般に，タスクが異なれば，データ領域も異なる．特別な場合を除き，同一のメモリ領域をアクセスする場合の競合(順序の保証など)を考慮する必要はない．しかし，一つのタスクを分割しているプロセス(正確にはスレッド)間では共通のメモリ領域(変数領域)をアクセスすることは珍しくない．というより，メモリ領域はほとんどが共通といってもよい．

マルチプロセッサ構成では，複数のプロセッサが互いにアクセス可能な共有メモリを用いるのが普通である．このような構成を共有メモリ型マルチプロセッサ(Shared Memory Multiprocessor)という．この共有

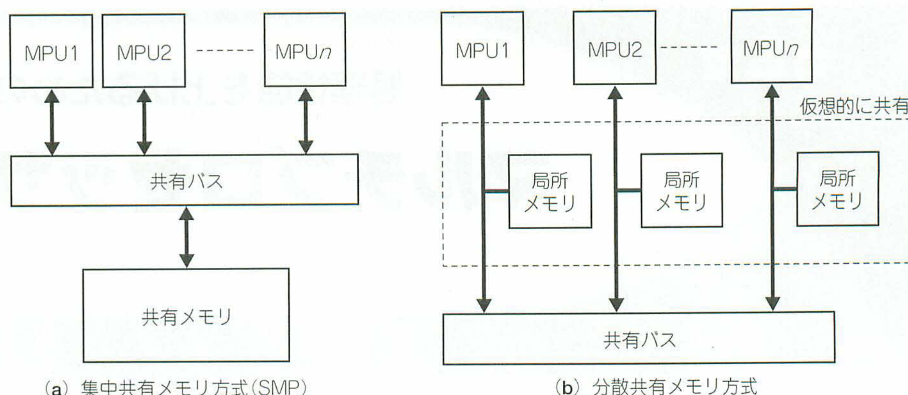


図1 マルチプロセッサの構成

メモリの構成方法によって，**図1**のように，マルチプロセッサの構成は2種類に分類できる．より一般的には，プロセッサと共有メモリを結合する方法は，共有バスとは限らず，相互結合網と呼ばれる．しかし，ここでは簡単のために，共有バスの場合を考える．

本書では共有メモリのあり方に注目して分類する．またマルチプロセッサには，密結合マルチプロセッサ(Tightly Coupled Multi-Processor = TCMP)と疎結合マルチプロセッサ(Loosely Coupled Multi-Processor = LCMP)という分類もある．複数のプロセッサが一つのOSと一つのメモリを共有する形態がTCMP，個々のプロセッサがそれぞれのメモリとOSをもち(つまり独立したコンピュータシステムを形成し)，入出力ポートなどを通じて通信(ネットワークや高速バスで結合)を行うのがLCMPである．大型計算機では，密結合マルチプロセッサを疎結合することで性能を向上させている．以下に説明するSMPはTCMPと同一視されることが多い．

▶ 集中共有メモリ方式

これは，プロセッサから共有メモリへのアクセス時間が一定という構成である．すべてのプロセッサが時間的空間的に対称なので，対称型マルチプロセッサ(SMP = Symmetric Multi-Processor)とも呼ばれる．SMPにおいて，複数のプロセッサは同等なものとして扱われる．また，共有メモリはすべてのプロセッサから同一のアドレスでアクセスできる．共有メモリが一つしかないので，SMPはUMA(Uniform Memory Architecture)とも呼ばれる．UMAとは，本来は異なる属性のメモリを同一のメモリ空間上に配置するアーキテクチャである．現在では，グラフィックス用のフレームバッファをメインメモリの一部として確保する方式を指すことが多い．

SMPの利点としては，各CPUが簡単にデータを共有でき，並列のプログラムが簡単に書けること，逐次処理のプログラムやプロセスを処理するのに優れている点が挙げられる．逆に弱点は，同時に一つのプロセッサしか共有メモリにアクセスできないので，CPUの数が増えるにしたがいアクセス権の調停が難しくなることである．しかし，その簡易性から，ほとんどすべてのマルチプロセッサではこの方式を採用している．このため，多数のCPUをマルチプロセッサ構成にすることが難しく，現実には4CPU程度が限界となっている．

キャッシュ内蔵のCPUをマルチプロセッサ構成で使用する場合，コヒーレンシの問題が発生する．後で説明するが，UMAでは，各CPUがアドレスバスを常に監視(スヌープ)して，自身のもつキャッシュ内のデータにそのアドレスが一致するかどうかチェックして，そこを無効化またはデータ更新することでコヒーレンシを維持する．

SMPの反意語として非対称型マルチプロセッサ(ASMP = Asymmetric Multi-Processor)というものもある．これは，次に説明する分散共有メモリ方式を示す場合もあるが，OSのカーネル用，ユーザープログラム用というようにCPUの役割を区別する方式を示す場合もある．SMPとは異なり，ASMPの定義ははっきりしていない．

▶ 分散共有メモリ方式

これは，プロセッサごとに局所メモリをもつ構成である．プログラムの実行には参照という局所性があるので，この構成なら共有バスの転送量を低減できる．プログラムを容易にするために，局所メモリに連続したアドレスを割り付け，論理的に単一の共有メモリとしてアクセスできるようにする．UMAと対応して，NUMA(Non-Uniform Memory Architecture)と呼ばれる．

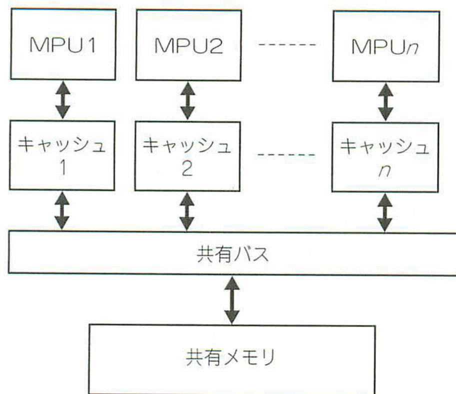


図2 マルチプロセッサとキャッシュ

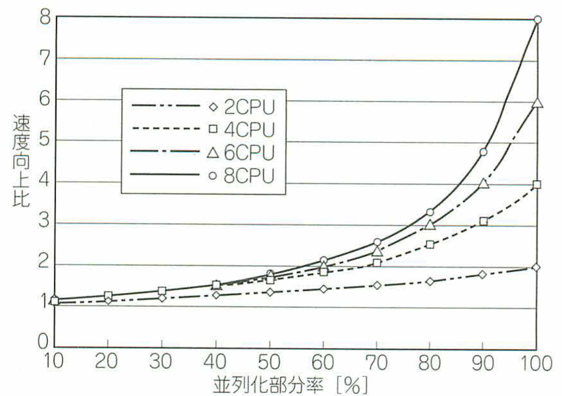


図3 アムダールの法則

バス速度，アクセス競合の点で，高並列マシンの集中共有メモリ方式は現実的でない．この場合，分散メモリ方式を採らざるをえない．現実としては，**図2**のように，SMP構成で各プロセッサにキャッシュをもたせる方式が一般的である．キャッシュはプログラムから意識されないため，SMPでありながら，バス速度，アクセスの競合の問題を解決できる．

NUMAにおいて，キャッシュのコヒーレンシを保証するシステムはccNUMA(Cache Coherent Non-Uniform Memory Architecture)と呼ばれる．たとえば，SunのEWSであるSunFire15Kでは，18組の4CPUからなるSMPモジュールをccNUMAで結合して性能を上げている．

NUMAでは，自分のメモリへのアクセス時間に比べて，他のCPUのメモリへのアクセスが遅くなる傾向があり，それが採用の妨げとなっていた．しかし，現在では技術革新により3～5倍かかっていたアクセス時間が平均で1.5倍，最悪でも3倍程度に縮まり，たいした問題ではなくなっている．

● 並列処理の割り当て

マルチプロセッサはOSの助けを借りて並列動作を行うことができる．CPUを複数並べただけでは性能向上は望めない．OSがプログラムをプロセスなりスレッドなりに分解して，各CPUに処理を割り当てて実行させるのである．

さて，SMPに対してOSがそれぞれのプロセッサに処理を割り当てる方法には2通りの方法がある．一つはプロセス単位での割り当て，もう一つはスレッド単位での割り当てである．

プロセスというのは一つのアプリケーションプログラムそのものであるから，これはアプリケーションプログラムごとに一つのプロセッサを割り当てるというイメージであろう．OSとしてはとくに特殊な処理は必要ない．この場合，各アプリケーションの実行が終了する時間は，それぞれのプログラムが一つのCPUで実行される場合と同じである．アプリケーションが複数ある場合には，それらが同時に実行されるので，すべてのアプリケーションの実行を終了する時間は早くなる．当然，アプリケーションが一つの場合には効果がない．この意味で，あるプロセスだけに限ってみれば，CPUが一つのときと実行速度になんら変わりはない．

スレッドというのは一つのプロセスを並列実行可能な部分に分割したもので，プロセスはいくつかのスレッドを寄せ集めたものである．たとえればこれは，複数のアプリケーションプログラムが複数のCPUで時分割に実行されるイメージである．スレッドの割り当ての運がよければ(？)，一つのアプリケーションの各部分が別個のCPUに割り当てられて実行されることになる．つまり，あるプロセスに限ってみれば，CPUがひとつの場合よりも高速に実行できることになる．

このように，マルチプロセッサ(SMP)において「処理が速い」というのには2通りの意味があるので注意が必要である．マルチプロセッサで速度を向上させようとしたらスレッド分割にせざるをえない．

ところで，マルチプロセッサシステムでCPUの数をどんどん増やしていけばそれだけ処理速度が速くなるのか，という疑問がある．じつはそうならないことがアムダールの法則(Amdahl's Law)によって示唆されている(**図3**)．これは，

改良後の実行時間＝影響されない部分の実行時間
＋(影響される部分の実行時間/改良の度合)

というもので，上の式で「影響」を「並列実行」，「改良」を「プロセッサの個数」に当てはめればよい．つ

まり，どのようなアプリケーションプログラムでも並列に実行できない部分が存在する(前後関係の依存性がある)ので，並列化できる部分しか高速化できないというものである．並列処理を念頭に置く場合，性能向上は次の式で表される．

速度向上比 $= 1/((1-P)+P/N)$

$=$ 旧実行時間/新実行時間

P：並列化部分率(元の実行時間のうち，並列処理可能な実行時間の割合)

N：並列化度(プロセッサの個数)

2 マルチプロセッサのキャッシュ制御

● マルチプロセッサのキャッシュ

マルチプロセッサにおけるキャッシュは単一プロセッサと比べると複雑である．共有メモリのコピーがそれぞれのキャッシュに存在するが，共有メモリと各キャッシュ間で一貫性(コヒーレンシ，またはコンシステンシという)を保証する必要があるからである．システムにおいてコヒーレンシが保たれている状態をコヒーレントという．コヒーレントにおいては，あるプロセッサが任意のメモリ領域(共有メモリや他のキャッシュ全体)をリードした場合，常に最新のデータがリードできることが保証されなければならない．これは，通常，CPU自体のハードウェアで保証される．

共有バスがバスネックにならないように，マルチプロセッサ構成ではライトバック方式のキャッシュを採用することが多い．この場合，各キャッシュにダーティなデータ(キャッシュだけで書き換えられていて主記憶の内容と一致しないデータ)があることを許すが，この場合でも他のキャッシュと矛盾しないようにキャッシュ全体の状態を管理する必要がある．これは，あるプロセッサのキャッシュ内容が書き換えられた場合に，そのことを他のキャッシュに通知することで実現する．この方式には次の2種類がある．

▶ ライトアップデート方式

あるキャッシュブロックの内容を書き換えた場合に，システム全体のキャッシュをヒットすれば，その最新のデータに更新する方式である．書き換えたブロックのアドレスとデータを，共有バスを通じて他のキャッシュに送る．これをアップデート要求という．アップデート要求を受けたキャッシュは，自身がそのデータを有していればそこを転送されてきたデータに変更する．

▶ ライトインバリデート方式

あるキャッシュブロックの内容を書き換えた場合に，自分以外のシステム全体のキャッシュがヒットすれば，そこを無効化する方式である．この場合，書き換えたブロックのアドレスを共有バスを通じて他のキャッシュに送る．これをインバリデート要求という．インバリデート要求を受けたキャッシュは，自身にそのデータを有していればそこを無効化する．

キャッシュへのライト時に共有バス上をデータが行き来するという点で，これらの方式はライトスルー方式と大差ないかもしれない．ライトスルー方式の場合でも，他のキャッシュにアドレスを送り，それがヒットすればそのブロックを無効化する必要があるのは同様である．このような状況で，ライトバック方式とライトスルー方式のどちらのキャッシュヒット率が高いのかを考えると，はっきりいってよくわからない．この二つでは，感覚的にはライトバック方式のほうがヒット率が高いとは思うのだが．それにも増して，ライトアップデート方式では，キャッシュを無効化する頻度が低い分，ヒット率が高いのは明らかである．ただし制御は複雑である．

ところで，キャッシュミス時にはキャッシュのリフィルが行われるが，コヒーレンシを保つために，コヒーレントリードというバスサイクルを発生する．コヒーレントリードの発生を共有バス上に発見すると，各キャッシュはバススヌープ(キャッシュ状態ののぞき見)を行う．

もし，キャッシュにヒットするダーティなブロックがあれば，そのデータが最新のデータなので，そのキャッシュがコヒーレントリードのデータを返さなければならない(これがオーナーである．詳細は後述)．キャッシュにヒットするクリーン(主記憶とデータが一致している状態)なブロックがあれば，そこを無効化してもよいのだが，キャッシュの有効活用のために，その状態を「共有(シェアード)」に変更するだけでデータはそのまま残しておく．

余談ではあるが，上記のキャッシュのスヌープ方式は共有メモリ方式で使用される．共有メモリを使用しない分散メモリ方式では，キャッシュのコヒーレンシを保つためにディレクトリ方式が使用される．これは，分散された各メモリごとに各メモリのコピーがどのプロセッサのキャッシュに存在するかをメモリ上のテーブルに記録しておく方式である．この場合，キャッシュをスヌープしなくても，ディレクトリの内容を参照

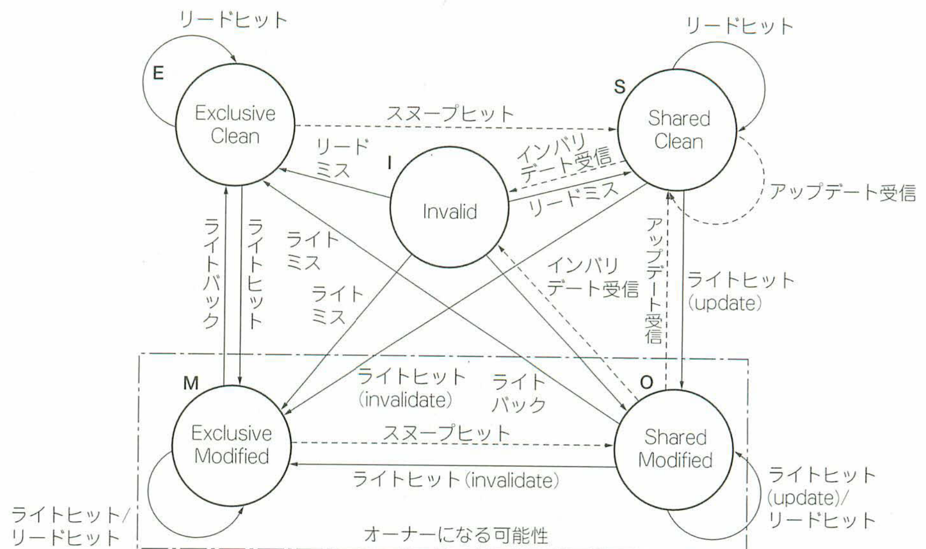


図4
MOESIモデルでのキャッシュの状態遷移

するだけでインバリデートやアップデート対象のキャッシュを特定できる．しかしこの方式では，主メモリ容量の増大にともなってディレクトリの容量が大きくなるという欠点がある．そこで，バススヌープ方式とディレクトリ方式を融合して効率よくコヒーレンシを保証する方法も研究されている．

● キャッシュの状態

ライトバック方式でコヒーレンシを保つために，キャッシュには次のような状態が定義されている．ある状態において，キャッシュに対してどのようなアクセスが行われるかによって，コヒーレンシを保つためにいろいろな挙動をする．

1) 無効(インバリッド)/有効(バリッド)
2) 不一致(ダーティ)/一致(クリーン)
3) 共有(シェアード)/排他(イクスクルーシブ)
4) 所有(オーナー)/非所有(ノンオーナー)

オーナーとは，最新で有効なデータブロックが複数存在している場合にデータを供給する責任のあるブロックのことである．ダーティなラインが存在しなければ主記憶，ダーティなラインが存在すればそのラインがオーナーである．その意味で，オーナーという特別な状態は不要である．

ところで，キャッシュの状態にどのようなものがあるかによって，キャッシュの方式にもいろいろな呼び名がある．ここでは，代表的なMOESI方式とMESI方式について説明する．なお，キャッシュデータの共有を許さないのはMEI方式と呼ばれる．これは，マルチプロセッサ機能を提供しない，通常のMPUのキャッシュ状態(ライトバック方式)を示すのに使用される．

▶MOESI

この方式では，キャッシュコヒーレンシをM(Modified)，O(Shared Modified)，E(Exclusive)，S(Shared)，I(Invalid)，という五つの状態で制御する．この方式での状態遷移を**図4**に示す．

● M(モディファイド)

ダーティともいう．主記憶と一致が取れていない状態．同じブロックはほかのキャッシュには存在しない．

● O(シェアードモディファイド)

主記憶と一致が取れていない状態．ほかのキャッシュに同じブロックが存在する．Oの由来はOwnerか？

● E(イクスクルーシブ)

主記憶と一致が取れている状態．同じブロックはほかのキャッシュには存在しない．

● S(シェアード)

主記憶と一致が取れている状態．ほかのキャッシュに同じブロックが存在する．

● I(インバリッド)

無効な状態．

▶MESI

この方式はMOESI方式からO状態を削除した方式である．シェアード状態のブロックにライトを行った場合，ライトインバリデート方式はインバリデート要求を発行し，自身はM状態に移行する．ライトアップデート方式はアップデート要求を発行して，他のキャッシュの内容も更新する．この方式での状態遷移を**図5**に示す．

MOSEI方式にしてもMESI方式にしても，ライト

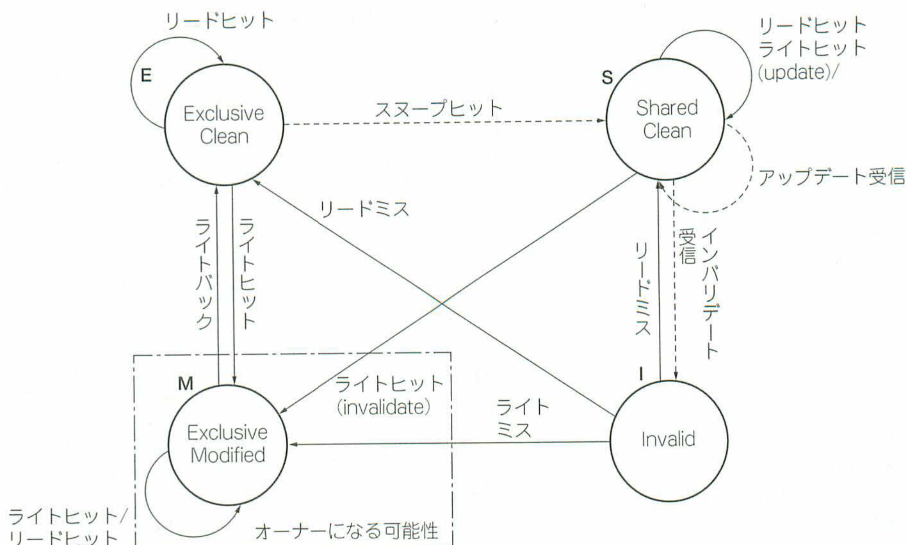


図5
MESIモデルでのキャッシュの状態遷移

インバリデート，ライトアップデートの両方の方式を使用可能である．それぞれシェアード状態のブロックにライトヒットした場合に，アップデート要求，インバリデート要求が発生する．

ところで，O状態を採用するとキャッシュの制御が複雑になるので，MESI方式のほうが多用されている．代表的な例では，MIPSのR4400ではMOESI方式とMESI方式を選択できたが，後継のR10000ではMESI方式のみとなっている．

● MESIプロトコルによる動作例

図6にMESIプロトコルによる動作例を示す．MESIプロトコルでは次のような状態遷移を行う．

1) リードヒット

ブロックが有効(M，E，S状態)ならリードするだけで状態は変化しない．

2) リードミス

他のキャッシュにヒットし，そこがM状態(最新，排他)であれば，そのブロックを主記憶にライトバックする．同時にそのデータを(コヒーレント)リード応答として要求元に返す．状態はS状態に遷移する(図6(a))．他のキャッシュにヒットし，そこがE状態またはS状態であれば，リード応答は主記憶によって返される．状態はS状態に遷移する〔図6(b)，図6(c)〕．他のキャッシュにヒットしない場合は，リード応答は主記憶によって返される．状態はE状態に遷移する〔図6(d)〕．

3) ライトヒット

ブロックがM状態ならそのまま上書きする．状態はM状態のままである．ブロックがS状態なら上書きを行い，状態をM状態にする．同時にインバリデート要求(ライトインバリデート方式を仮定)を発行し，他のキャッシュにヒットすれば，そのブロックを無効化(I状態)する〔図6(e)〕．ブロックがE状態なら上書きを行い，状態をM状態にする．インバリデート要求は発行しない〔図6(f)〕．

4) ライトミス

2)のリードミスと同じ動作を行った後，3)のライトヒット動作を行うのと等しい．

3 プロセス間の相互通信の方法

● プロセス間通信の必要性

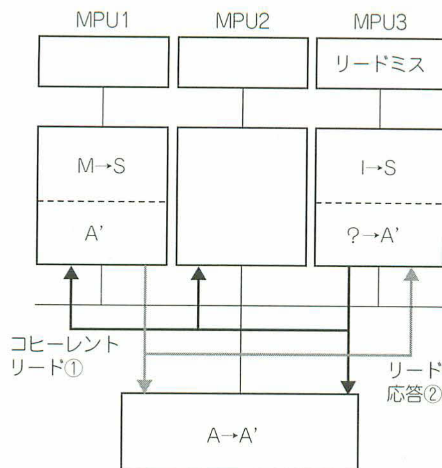
マルチプロセッサシステムは互いにほとんど独立して動く多数のプロセスの集まりである．独立しているとはいえ，各プロセスは相互に直接・間接に連絡を取り合う必要が生じる．この場合，自由に動いているプロセスの動きをあるタイミングでそろえることを，同期を取るという．プロセスが連絡を取り合うには同期が必要で，連絡は次の二つの場合に必要となる．

1) プロセス間の協力のため

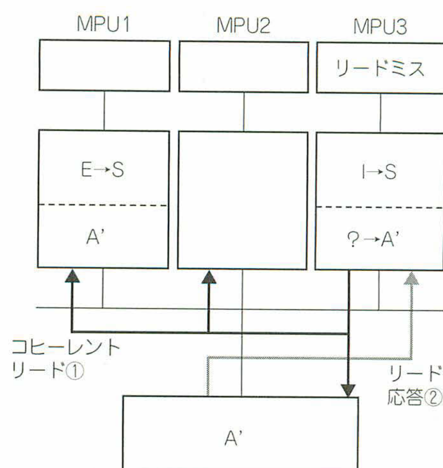
あるプロセスがほかのプロセスに仕事を依頼する場合，依頼したいという信号と依頼の内容を示すメッセージを送受するしくみが必要である．これはプロセス間の待ち合わせであり，条件同期という．

2) プロセス間の資源の競合を解決するため

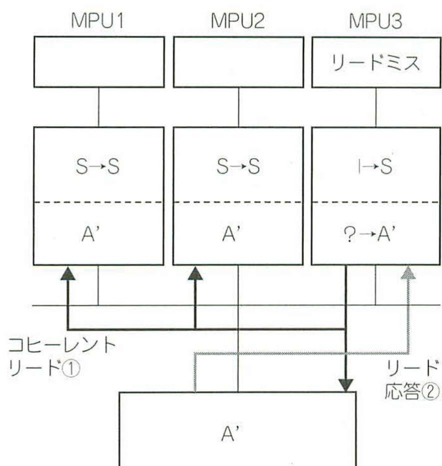
各プロセスで共有する資源は，たかだか一つのプロセスでしか利用できないような排他制御が必要である．これを相互排除という．



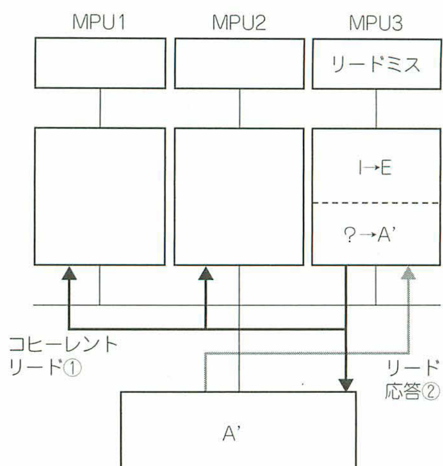
（a）リードミスで，他に同じブロックのコピー（Modified）がある場合



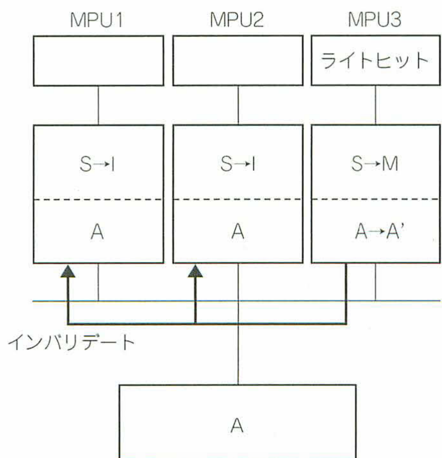
（b）リードミスで，他に同じブロックのコピー（Exclusive）がある場合



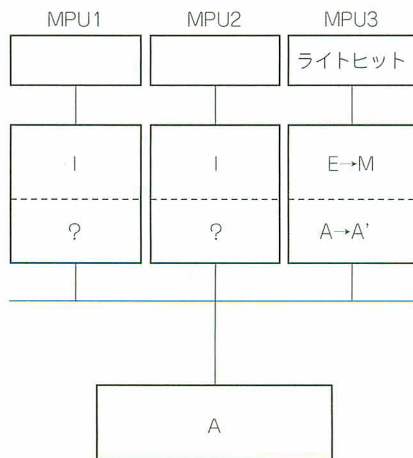
（c）リードミスで，他に同じブロックのコピー（Shared）がある場合



（d）リードミスで，他に同じブロックのコピーがない場合



（e）Shared状態でライトヒット



（f）Exclusive状態でライトヒット

図6　MESIプロトコルによる動作例

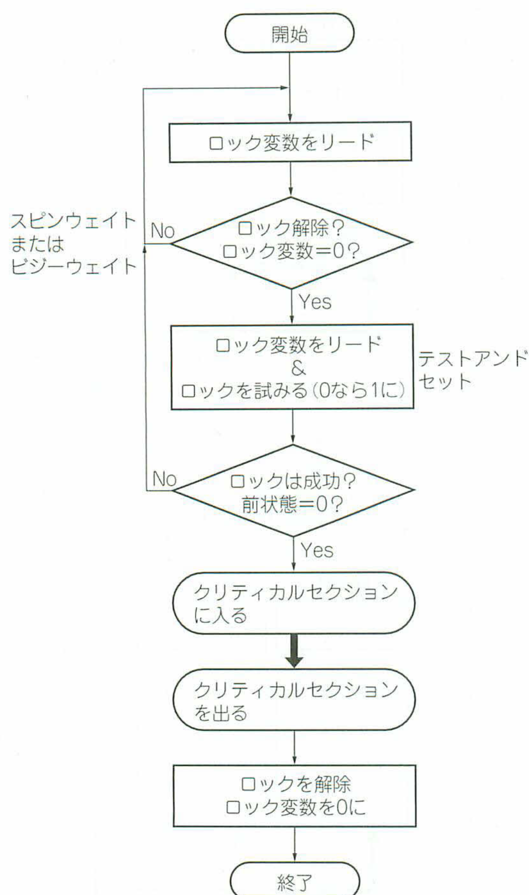


図7　スピンロック方式

● 同期のためのソフトウェア処理

相互排除の問題をまず考える．マルチプロセッサシステムでは，共有バスが主記憶にアクセスする唯一のアクセス手段である．メモリの排他制御は，バスの使用権を獲得したプロセッサがメモリアクセスを独占し，ほかを締め出せばよい．この処理は不可分（アトミック）なスワップ命令で簡単に実現できる．

不可分とは，あるプロセスがメモリにアクセスしている間にほかのプロセッサからのメモリアクセスがないことをいう．実際には（バス）ロックという端子を活性化して外部回路に通知し，ハードウェア的にほかのプロセッサのアクセスを禁止する．

このため各共有資源に対し，ロック変数を用意する．ロック変数が0の場合は，それに対応する資源が未使用，0でない場合は，それに対する資源が使用中であることを示す．プロセスはロック変数に0以外の値を早い者勝ちで書き込むことで，その資源を使用する権利をもつ．

それを実現するもっとも単純な処理が，テストアンドセットである．これは，ロック変数をリードしその値が0ならそこに1を書き込む．ロック変数が0でなければ何もしない．ロック変数の前の値が結果として返されるので，それが0ならロックが成功したことを示す．0でなければロックは失敗である（ほかのプロセスが使用中）．**図7**にテストアンドセットを利用した相互排除を示す．ここで，ロック変数が0になるのを待つループをスピンウエイト，またはビジーウエイトと呼ぶ．そのため，この方式をスピンロック方式と呼ぶ．

スピンロック方式は，他のプロセスがクリティカルセクション（共有資源を独占できる時間的空間的領域）を実行中は，同じ資源を使用しようとするプロセスがループしながら待つ点が非効率である．スピンウエイトを行わない同期機構としてセマフォがある．

セマフォとは，鉄道で用いられる腕木式信号機のことである．セマフォの実体は一種の共有フラグで，同期をとるプログラムどうしがこのフラグに注目し，フラグの変化に応じて処理を行って同期を実現する．構造化プログラミングの提唱者であるEdsger Wybe Dijkstraは，セマフォに対するP操作，V操作というものを定義した．P操作は事象の待ち合わせ，V操作は事象の発生（およびメッセージ）の通知を行う．セマフォを実現する操作はフェッチアンドアッドと呼ばれる．これはメモリに対する任意の数（負の数も可）の加算を不可分に行う．

P操作，V操作のP，Vという名称の由来はDijkstraの母国語のオランダ語にある．Pは「prolagen」という「proberen（試みる）」と「verlagen（減じる）」の合成語が語源である．また，Pには「passeren（渡す）」という意味も含まれているという．Vは「verhogen（増やす）」または「vrygeven（解放する）」が語源である．由来を知っていれば，以下に説明するそれぞれの動作も理解しやすいであろう．

セマフォiは，共有資源iについて定義されており，負の値も取るカウンタSiと待ち行列Qiからなる．これに対するP操作，V操作の流れを**図8**に示す．簡単にいえば，P操作で資源が得られない場合は，そのP操作を行ったプロセスは待機状態となり，Qiに格納され，後で起動されるのを待つ．これをサスペンド状態という．V操作は待ち行列Qiの中にあるプロセスを起動する．

相互排除はセマフォのカウンタの初期値を1に設定

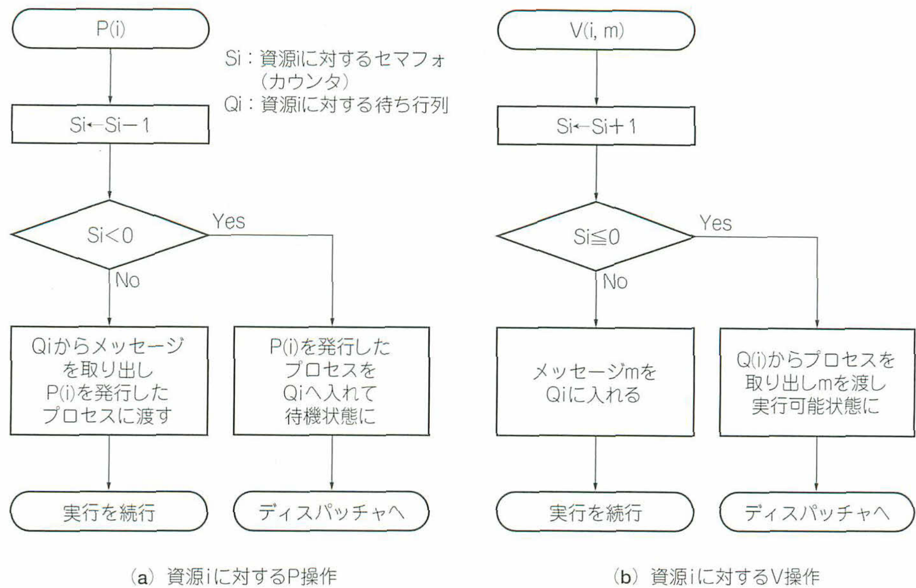


図8
P操作とV操作

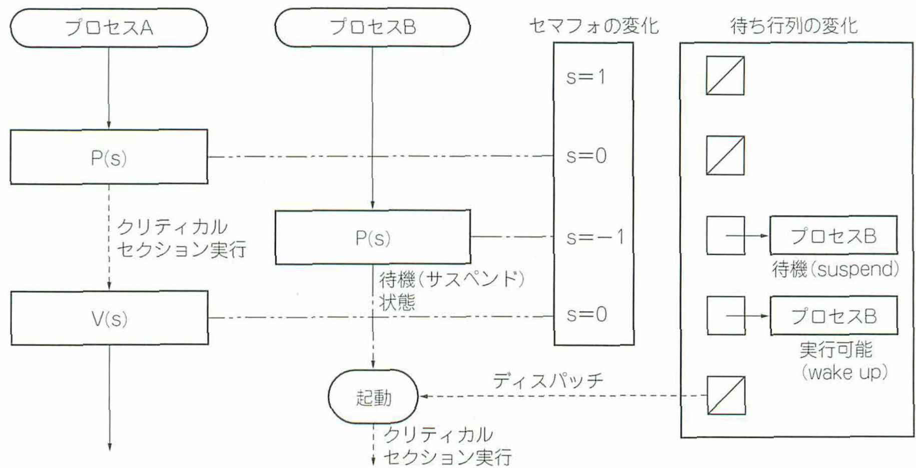


図9
サスペンドロック方式

することで実現できる．**図9**に示すように，P操作でロックを行い，V操作でロック解除を行える．この場合，スピンウエイトは存在せず，プロセスをサスペンド状態にすることで待ち合わせを行う．この意味から，セマフォを利用した相互排除をサスペンドロックと呼ぶ．相互排除では，一つのセマフォに対するP操作とV操作は同一のプロセス（プロセッサ）によって発行される．

セマフォはカウンタの初期値を0に設定することで，条件同期にも使用できる．**図10**に示すように，二つのプロセス間で，同じ資源に対するP操作とV操作をそれぞれ発行することで待ち合わせができる．この場合は，一つのセマフォに対するP操作とV操作は別個のプロセス（プロセッサ）によって発行される．

セマフォを使ったプロセス間の通信（待ち合わせ）を利用する代表例として，生産者・消費者問題（producer-consumer problem）がある．これは**図11**に示すように，生産者プロセスAが生産を行って次々に結果を主記憶に確保されたバッファに書き出していく．同時に，消費者プロセスBはバッファから結果を取り出して何らかの処理を行い，不要になったバッファを返却するというモデルである．このとき，生産と消費の速度が同じとは限らないので，バッファが一杯なのに結果を書き込んだり，バッファが空なのに結果を取り出したりしないような考慮が必要である．それをどうすれば実現できるのかが問題である．この問題を，セマフォを用いて解くと次のようになる．

- 生産者：P1：生産をする．
  P(e)で空きバッファを要求する．

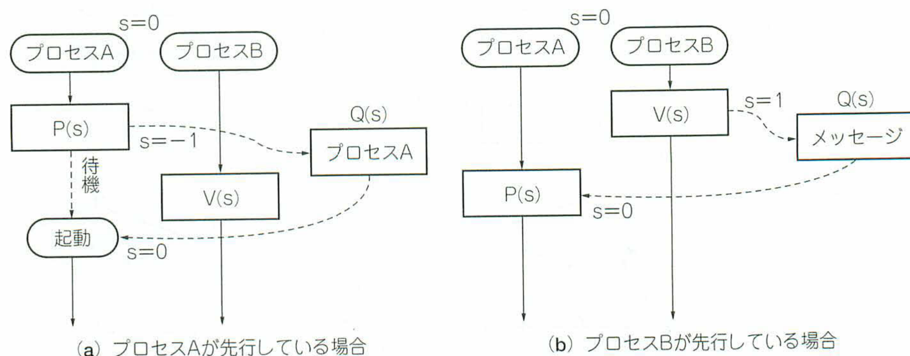


図10
条件同期(待ち合わせ)

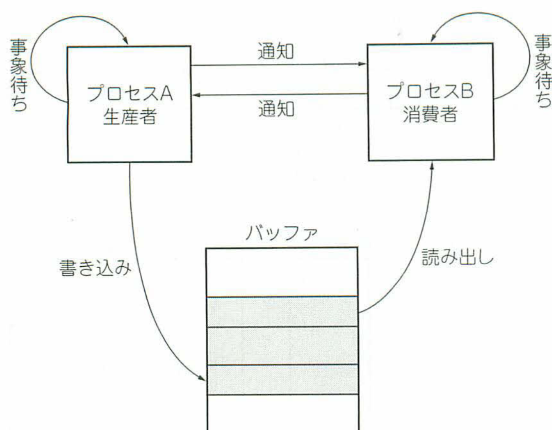


図11　生産者・消費者問題

空きバッファに結果を格納する．
V(r)でバッファに結果が入ったことを通知する．
GOTO P1

- 消費者：C1：P(r)で結果入りバッファを要求する．
  バッファから取り出して処理する．
  V(e)でバッファを消費したことを通知する．
  GOTO C1

ここで，eは空きバッファを制御するセマフォ，rは結果の入ったバッファを制御するセマフォである．eの初期値はバッファの段数，rの初期値は0である．

● 同期のためのハードウェア(命令セット)

各MPUのアーキテクチャでは，これまで説明してきた，不可分な(アトミックな)アクセスを実現するための最小限の手段を命令として提供している．これらをアトミック命令という．以下に，これらアトミック命令について述べる．

不可分なアクセスの基本は，データをリードして変更してライトする，といったリードモディファイライト(Read Modify Write = RMWと略す)が基本になる．CISCにおいてはRMWはポピュラな動作なのでそれを排他的に実行できればよい．具体的には，MPUの外部端子のLOCK信号(MC680x0ではRMC端子)をアクティブにして，それがアクティブの間は外部回路でほかのプロセッサからのバスへのアクセスを禁止することである．LOCK端子でバスの競合を防ぐことが目的なので，バスサイクルが常に共有バスに発生する必要がある．つまり，ロック変数は非キャッシュ領域に配置する必要がある．あるいはキャッシュ領域に置く場合は，そこがキャッシングされないようにモニタしなければならない．しかし，実際には，プログラマがそれを考慮する必要はない．

たとえば，MC680x0アーキテクチャではTAS命令，CAS命令，CAS2命令のRMWアクセスにおいては，ハードウェアによって自動的にデータキャッシュが必ずミスするようになっている．このとき，バースト転送は行われないが，キャッシュの内容は更新される．

x86アーキテクチャにおいて，PentiumまでのP5アーキテクチャではLOCK端子がアクティブなバスサイクルはキャッシングされない．つまり，最初がキャッシュミスを発生する領域(無効状態)では永久にキャッシングされず，常にバスサイクルが発生するわけである．PentiumPro以降のP6アーキテクチャでは，キャッシュ領域に対するアトミック操作はLOCK端子をアクティブにはせず，キャッシュロックという状態になる．これは，いわゆるキャッシュを追い出さないキャッシュロックとは異なるので注意が必要である．これはほかのプロセッサが同じキャッシュラインのライトバックを行うときにインバリデートされてコヒーレンシを保つものらしい(詳細はよくわからないが，キャッシュロック領域は自動的にバススヌープ対象となるということなのだろう)．非キャッシュ領域に対するアクセスではP5と同様にLOCK端子をアク

リスト1
テストアンドセット

```
Test-and-Set(address, bit_position)
{
    Lock();                                     ロック端子をアクティブ
    temp ← Memory[address].bit_position;        共有変数から指定ビットをリード
    if(temp == 0) then
    Memory[address].bit_position ← 1;           リード値が0なら1をセット
    endif
    UnLock();                                   ロック端子をインアクティブ
    Condition_Code ← temp;                      前の値を返す
```

リスト2
コンペアアンドスワップ

```
Compare-and-Swap(address, cmp_data, new_data)
{
    Lock();                             ロック端子をアクティブ
    temp ← Memory[address];             共有変数をリード
    if(temp == cmp_data)then
      Memory[address] ← new_data;       比較値と等しければ新しい値をセット
    endif
    UnLock();                           ロック端子をインアクティブ
    cmp_data ← temp;                    前の値を返す
}
```

リスト3
フェッチアンドアッド

```
Fetch-and-Add(address, e)
{
    Lock();                             ロック端子をアクティブ
    temp ← Memory[address];             共有変数をリード
    Memory[address] ← temp + e;         値eを加える
    UnLock();                           ロック端子をインアクティブ
    return temp;                        前の値を返す
}
```

ティブにしたバスサイクルが発生する．資料には「P6以降はバスエージェント(チップセット)の制御により，ほかのプロセッサがバスをアクセスするのを制限する」とあるので，LOCKプリフィックスはあまり利用されないらしい．キャッシュへの排他アクセスはMESIプロトコルでコヒーレンシを保証する．

一方，RISCではロードとストアを別個の命令で実現するので，1命令でのRMW操作は不可能である．これには，RMW専用のロード命令とストア命令を用意して対応する．後で説明するRISC方式の場合，ロック変数はキャッシュのコヒーレンシを利用するので，キャッシュ領域に置かなければならない．ただし，i860や88000などの初期のRISCでは，CISCと同様のRMWでアトミック操作を行う命令をもっていたようである．

▶ テストアンドセット(Test And Set)

通常，1ビットのロック変数に対するRMW命令である．共有変数のアドレスとビット位置を入力として次のように定義される．ビット単位ではなくワード単位にRMWを行う場合もある(**リスト1**)．

この操作は，680x0アーキテクチャでは，

```
TAS   <ea>
```

として提供されている．ただし，TASではデスティネーションの元の値にかかわらず，必ず1(デスティネーションのビット7)をセットする．元の値は条件フラグに反映される．元の値が0であろうが(ロックされていない場合)，0でなかろうが(すでにロックされている場合)，その結果は必ず0以外となるので，この動作で十分である．

▶ コンペアアンドスワップ(Compare And Swap)

この操作はテストアンドセットの発展形である．比較する対象が0ではなく任意の値となる．具体的には，1回前の共有変数の値を記憶しておき，それと今回の値と比較し，値が同じならば，ほかのプロセッサからのアクセスがないとみなされるので，新しい値を共有変数にセットする(**リスト2**)．

この操作は，680x0アーキテクチャでは，

```
CAS.{B, W, L}  Dc, Du, <ea>
```

として提供されている．なお，マニュアルでは同期操作のほかにリンクリストのポインタを付け替える操作がCAS命令の代表例として挙げられている．

▶ フェッチアンドアッド(Fetch And Add)

この操作は共有メモリに対する不可分な加減算を実現する．セマフォに対するP操作とV操作の一部を簡単に実現できる(**リスト3**)．

この操作を命令として提供するアーキテクチャは少ない．しかし，これはコンペアアンドスワップ命令で実現できる．たとえば，680x0ではCAS命令を使用

リスト4
ロードリンクト/ストアコンディショナル

```
Load-Linked(address)
{
    temp ← Memory[address];     共有変数をリード
    LLbit ← 1;                  LLbitを1に設定
}

Store-Conditional(address, temp)
{
    if(LLbit == 1)then
      Memory[address] ← temp;   LLbitが1なら値を共有変数にライト
    endif
    temp ← LLbit;               LLbitを返す
}
```

して次のように記述する.

```
    move.w 0(a0), d0
loop:
    move.w d0, d1
    add.w  1, d1
    cas.w  d0, d1,0(a0)
    bne    loop
```

この例はスピンロックになるのが気に入らないが，まあよしとしよう.

▶ロードリンクト(Load Linked)，ストアコンディショナル(Store Conditional)

これはMIPSアーキテクチャで導入された命令である．MIPSの産みの親であるHennessy教授によると，初期MIPSのアーキテクトであるEarl Killianの発案という．LL(Load Linked)命令とSC(Store Conditional)命令はキャッシュ領域への不可分なRMWのための基本操作を提供する．この機構はMPU内部のLLbitという特別な資源を介して，共有変数のリードを行ったという状態を記憶しておく(**リスト4**).

つまり，あるプロセッサでのLL命令の実行それ自体では何も起こらず，同じ共有変数に対するほかのプロセッサからのSC命令の実行は失敗する．つまり，ライトは行われない．同じプロセッサからのSC命令には成功する場合と失敗する場合がある．SC命令のライトが行われる場合は不可分なRMWが成功したことを示す．SC命令が成功したか失敗したかはソースオペランドに格納される値(LLbitそのもの)が1であるか0であるかをテストすることで知ることができる．なお，LL命令とSC命令の実行の間に，次のいずれかの事象が発生するとSCは失敗する.

- 指定したメモリ領域への，他のプロセッサからのコヒーレントライト(インバリデート要求，またはアップデート要求が発行される)が行われた場合.
- LL/SCを実行するプロセッサに割り込み/例外が発生した場合．この実装は割り込みからの復帰命令(ERET)の実行によってLLbitをクリアするようになっていることが多い.

つまり，このような事象が発生するとLLbitが0にクリアされるようになっている．このようなRMWの実現方法は特異ではない．たとえば，PowerPCにおいても同期を実現するために，

```
LWARX (Load Word and Reserve Indexed)
STWCX (Store Word Conditional Indexed)
```

というロード命令とストア命令の組を提供している．PowerPCではLLbitの代わりにリザベーション(Reservation)というMPUの内部資源を利用する．考え方はMIPSとまったく同じである．これらの命令を使って，フェッチアンドアッドを実現すると次のようになる.

```
  top
    lwarx r9=i(r2)      # iをロードして予約
    ai r9=r9+1          # iをインクリメント
    stwcx i(r2)=r9      # 条件付きストア
    bc top, cr0         # ストア失敗は分岐
```

▶バスロックプリフィックス

x86アーキテクチャでは不可分なアクセスを実現するための専用命令はない．しかし，LOCKというプリフィックスを提供し，(基本的にはすべての)命令のRMWのメモリアクセスをバスロック付きで行える．ただし，アーキテキクチャでLOCKプリフィックスの動作を保証しているのは**表1**の命令とアドレシングの組み合わせのみである.

マニュアルによると，この組み合わせに違反した場合は例外が発生するとなっている．なお，XCHG命令に関してはLOCKプリフィックスなしでもバスロック信号をアクティブにする．Pentium以降で追加されたXADD(exchange and add)，CMPXCHG(compare and exchange)，CMPXCHG8B(Compare and exchange 8 bytes)は，LOCKプリフィックスと組で使用されることを前提とした命令である(単独でも使えるが).

x86アーキテクチャにおいてLOCKプリフィックスを使用すれば，テストアンドセット，フェッチアンドアッドの実現は簡単で，それぞれ次のようになる．

```
LOCK:BTS   mem, imm
LOCK:ADD   mem, imm
```

なんのことはない．ビットセット命令や加算命令にLOCKプリフィックスを付加しただけである．

▶ 各アーキテクチャのアトミック命令

表2にMPUの各アーキテクチャでどのようなアトミック命令が定義されているのかをまとめておく．これをみると，テストアンドセットでお茶を濁している(?)ものが大半である．最小限の機能さえ提供しておけばソフトウェアが何とか頑張ってくれるだろうという，いかにもRISC的な発想なのかもしれない．

4 マルチプロセッサの構造

● マルチコアプロセッサのいろいろ

性能向上のため，2～4個のCPUコアをマルチプロセッサ構成にして1チップに集積することが最近の流行である．とくに，性能が要求されるネットワーク処理用途のプロセッサに多い．そもそも，このような1チップマルチプロセッサの発想は昔から存在する．有名なところでは，Intelが1989年に2000年のマイクロプロセッサの姿を予想したMicro2000であろう．これは，大容量のキャッシュの周りに4個の汎用プロセッサ，2個のベクトルプロセッサ，1個のグラフィック処理プロセッサを1チップに集積している．2000年時点で，現実はIntelの予想に追いついていないが，その姿は見え始めている．

▶ IBM Power4

IBMは1999年のMicroprocessor Forumで1チップに二つのCPUを集積するPower4を発表した．各CPUは(瞬間的な)5ウェイスーパースカラで，1GHz以上の周波数で動作する．さらに，Power4を4個搭載したMCM(Multi-Chip Module)を作成し，それを4個SMP結合して32ウェイccNUMAを構成して高速化を図るという．2001年10月にIBMはPower4を採用したeServer p690を発表した．

これについては詳細を後述する．

▶ NEC MP98

2000年のISSCC(世界的権威のある国際固体素子回路学会)では，NECがV800プロセッサを1チップに4個集積したMP98を発表している(**図12**)．これは1GIPS(Giga Instructions per Second)の性能を達成するために，各プロセッサの性能を1/4の250MIPS程度として，消費電力を1Wに抑えている点が興味深い．各プロセッサは2ウェイスーパースカラなので，理論上は125MHz動作で大丈夫だ．MP98は低消費電力を実現するため，この分野にありがちなネットワーク処理よりも，モバイル端末を用いたインターネットへの接続，自動通訳，画像処理などを目的としている．専用並列化コンパイラによる音声認識プログラムを用いたベンチマークでは，1プロセッサ動作時と比較して，約3倍の性能向上を達成したという．なお，MP98はアーキテクチャをARMに変更して世界進出をねらっているらしい．

事実，NECエレクトロニクスとARM社は，2003年10月20日に業務提携を発表し，協同してARM11

表1 LOCKプリフィックスの動作を保証している命令(x86)

ADC, ADD, AND, BT	mem, reg/imm
BTS, BTR, BTC, OR	mem, reg/imm
SBB, SUB, XOR	mem, reg/imm
XCHG	mem, reg
XCHG	reg, mem
DEC, INC, NEG, NOT	mem
XADD, CMPXCHG	mem, reg
CMPXCHG8B	mem

表2 各アーキテクチャのアトミック命令

方式＼アーキテクチャ	x86	680x0	MIPS	ARM	SH	PowerPC	SPARC	Alpha
TEST and SET	LOCK BTS	TAS		SWAP	TAS		LDSTUB SWAP	PALコードで実現
COMPARE and SWAP	LOCK CMPXCHG	CAS					CASA* CASXA*	PALコードで実現
FETCH and ADD	LOCK ADD							PALコードで実現
LOAD LINKED STORE CONDITIONAL			LL SC	LDREX** STREX**		LWARX STWCX		

*）SPARC V9以降定義　　**）ARM v6以降定義

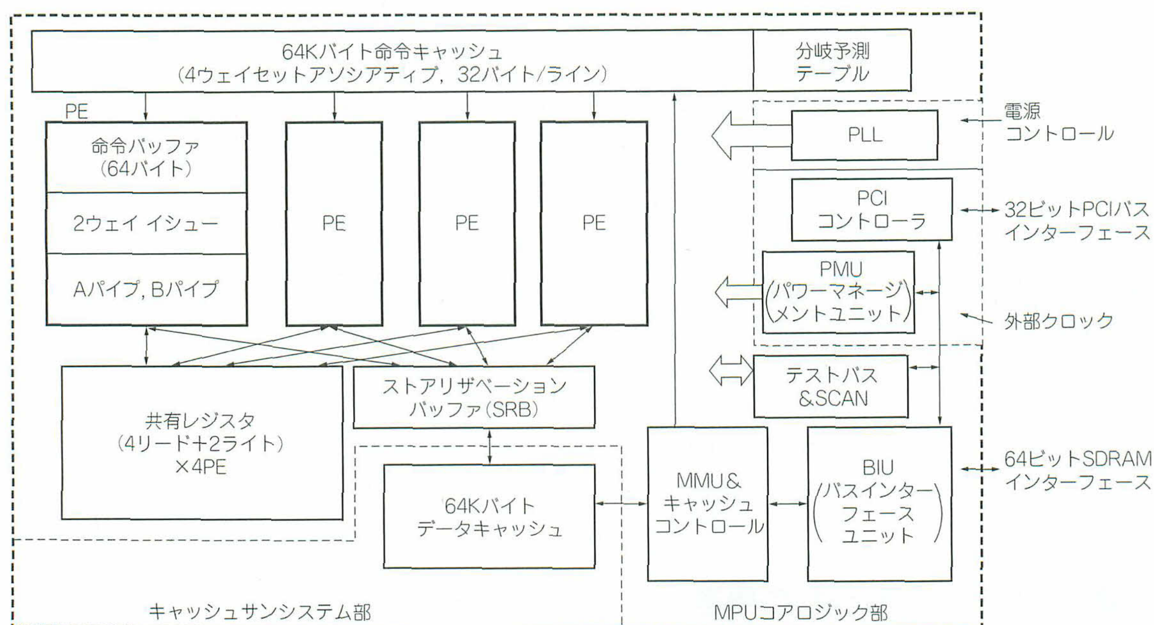


図12　NEC MP98のブロック図

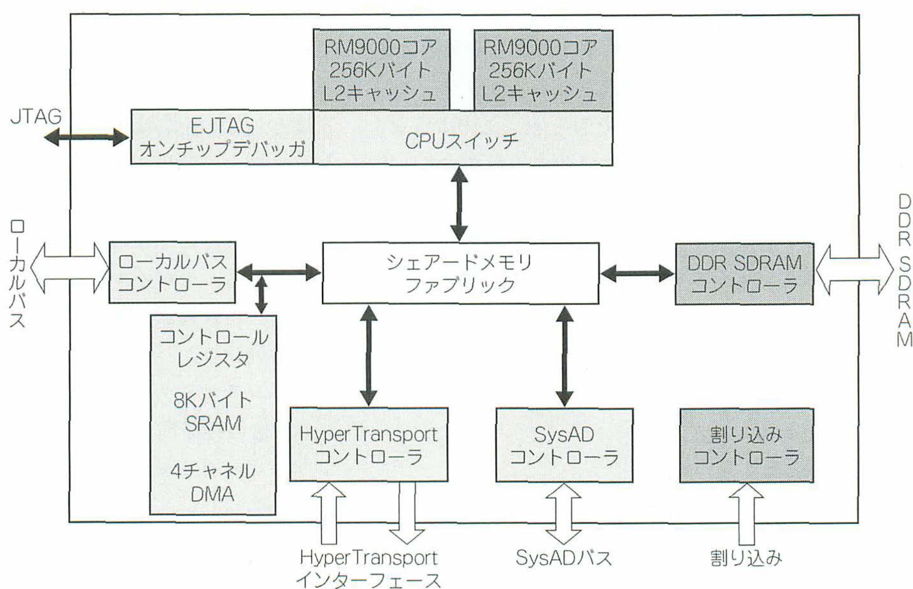


図13
PMC-Sierra RM9000x2
のブロック図

をマルチプロセッサ化することとなった．NECエレクトロニクスはSMP技術を提供するとなっているが，これはMP98で培われたものであることは想像に難くない．

▶**PMC-Sierra RM9000x2**

2001年のEmbedded Processor ForumではPMC-Sierra社(旧QED)がRM9000x2を発表した(**図13**)．RM9000x2は，その名称のとおり，64ビットMIPSプロセッサであるRM9000を2個1チップに集積したMPUである．RM9000自体は1GHzで動作する7ステージパイプラインの2ウェイスーパースカラである．パイプラインを従来の5ステージから7ステージに変更したことで，同じ製造プロセスであっても，67%の高速化が実現できる．1次キャッシュは命令，データ，それぞれ16Kバイトであり，2次キャッシュは256Kバイトのユニファイドキャッシュである．2次キャッシュはMOESIプロトコルを採用し，ヒット率の向上を図っている．また，プロセス間同期のために，従来の

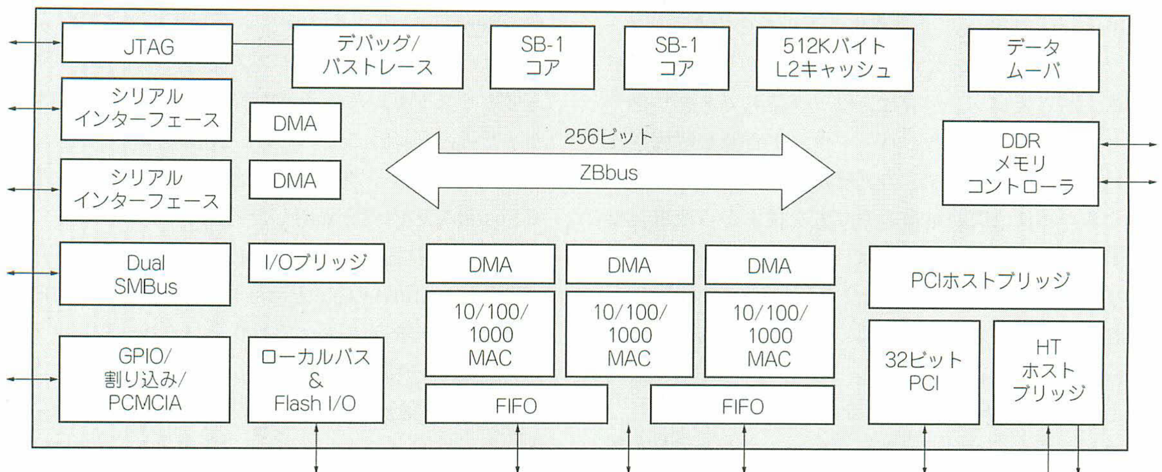


図14　Broadcom BCM1250のブロック図

LL/SC命令のほかに，不可分にセット/クリア可能なハードウェアレジスタと不可分なカウンタを提供する．

▶ **Broadcom BCM1250**

Broadcom社も，SiByte製のMIPS64プロセッサであるSB-1を2個集積するBCM1250(**図14**)を発表している．1GHzで動作し，そのときの性能は4400MIPSという．しかも，800MHz動作時の消費電力が8～10Wと，このクラスのMPUとしては非常に小さいのが特徴である．マルチプロセッサバスや512Kバイトの2次キャッシュを内蔵するなど，RM9000x2への対抗意識がみてとれる．

▶ **その他**

このほかに，HPのPA-8700の後継機種であるMakoやSGIのMIPSプロセッサの最上位機種であるR18000も二つのCPUを1チップに集積するデュアルプロセッサになるといわれている．デュアルプロセッサはこのように性能向上のかぎとなる技術になりつつあり，最近ではチップマルチプロセッサ(CMP：Chip Multi-Processor)という用語すら生まれている．

十数年前から，1チップマルチプロセッサはMPUの最終形態とされてきた．これらを採用するMPUがちらほらと現れてきたということは，MPUの性能向上がそろそろ限界に達しつつあることを示しているのかもしれない．SunもデュアルプロセッサのUltra SPARC Ⅳを2003年下半期に発表している．

余談ではあるが，RM9000x2の消費電力はかなり大きく(といっても，20W程度だと思われるが)，組み込み用途には向かないらしい．そこで，PMC-SierraはRM9000x1というCPUを1個のみ有効にしたMPUを積極的に販売しているようだ．CMPは組み込み用途に本当に必要なのかどうかを考えさせるエピソードである．

● **マルチスレッド**

マルチプロセッサは複数のMPUでプログラムを実行する方式であるが，一つのプロセッサをマルチプロセッサのように見せかける技術をもつ．それがマルチスレッドである．ソフトウェア(OS)からは二つ以上のプロセッサがあるように見えるので，従来のマルチプロセッサ用の処理をそのまま適用できるのがミソである．

マルチスレッドとは，MPUのレジスタファイルの組を複数用意し，複数のスレッドを同時実行させるしくみである．1チップマルチプロセッサとの違いは，キャッシュや演算器などのハードウェア資源が共通であるという点である．当然，複数のスレッド間でハードウェア資源の取り合いが生じるので，純然たるマルチプロセッサよりも効率は悪い．しかし，ハードウェア規模は小さくなる．

マルチスレッドでは，キャッシュミスなどの待ち時間を利用して別のスレッドを実行する．こうすることにより，プロセッサの待ち時間を最小に抑えることができる．もっとも，CPUコア単体では，命令ストリーミングやノンブロックキャッシュにより，キャッシュミスによるペナルティは少なくなっている．それよりも，周辺ユニットとの間のI/Oやメモリバスの割り当て待ちで浪費する時間のほうが多い．いずれにせよ，MPUのパイプラインはどこかで資源の待ち合わせが必要になる．その待ち時間を他のスレッドに割り当てるわけである．要するに，マルチスレッドは，マルチタスクOSがソフトウェアで実現している処理を

MPUのハードウェアで実現するものなのである．

マルチスレッドの欠点としてスピンロックの問題がある．あるスレッドがスピンロック待ちでメモリをアクセスし続けていると，ハードウェア資源が解放されないので，別のスレッドの実行ができない．これを回避するためには同期処理のやり方を根本から変更しなければならない．たとえば，割り込みが来るまでCPU(正確にはそのスレッド)を停止させる専用命令を利用するなどの方法が必要である．この制限はかなり致命的であり，マルチスレッド対応のMPUの性能を上げられない理由の一つとなっているようだ．

マルチスレッドには，キャッシュミスなどで長時間待つことが明らかな場合に，別プログラムの実行に切り替えるVertical Multi Threadingという方法と，常に複数のプログラムから命令を取り出し，ごちゃまぜに実行するSimultaneous Multi Threadingという方法がある．ハードウェア構造はVertical Multi Threadingのほうが簡単であるが，命令の実行効率はSimultaneous Multi Threadingのほうが勝っている．

マルチスレッド技術をプロセッサの製品ロードマップに載せているメーカーは多い．とくにAlphaを有していたCompaqとPowerPCのIBMが最先端を行っている．しかし，現実にはハードウェア構造を単純にするために1チップマルチプロセッサのほうが好まれる傾向にあるようだ．マルチスレッドにしろ，1チップマルチプロセッサにしろ，利用できるハードウェアの量から考えると当面の並列度は2～4である．

● ハイパースレッド(Intelのマルチスレッド)

2001年8月のIDFにおいてIntelは2002年に登場のPentium4(Foster)で2スレッドのSimultaneous Multi Threadingを実装予定と発表した．実際にデモも行われた(経験的にIDFのデモは当てにならないが)．この場合の性能向上は30％程度だという．Intelはこのマルチスレッディングをハイパースレッディング(Hyper-Threading)と呼んでいる．Intelの方式は命令デコード時にスレッドを切り替えるSimultaneous Multi Threading方式らしい．Xeonは基本的にSMP構成のデュアルプロセッサなので，それぞれのプロセッサがハイパースレッディングを行うと，ソフトウェアからは四つのマルチプロセッサに見える．

このマルチスレッディングはCompaqから譲渡されたAlpha技術の産物とみる専門家も多いようだ．噂によれば，じつはPentium4は最初のWillametteですでにハイパースレッディング機能を内蔵していたという．Pentium4のパイプラインが空きすぎてIPCが低いのはハイパースレッディング機構に備えてと推測するアナリストもいる．それが特許の問題で公表できなかったのだということらしい．Alpha技術とともにマルチスレッディングの特許を獲得したIntelが，晴れてFosterでの内蔵を表明したのだとか．

この問題の答えはIntel自身によって明確にされている．Pentium4のNetBurstは約10年という期間を見据えたアーキテクチャであり，ハイパースレッディングもその一環である．Willametteではすでに実装されている(完動するか否かは不明)が実行できないようになっているらしい．FosterとWillametteは同一のダイ(キャッシュサイズは異なる)だという．Alpha技術との関連はアプローチが異なり無関係だそうだ．

Fosterの後継で2002年にOEM評価用に出荷されたPrestoniaコアのPentium4でのハイパースレッディングのベンチマークによる性能評価結果が某サイトに掲載されていた．それによるとハイパースレッディングを許可すると平均的に39％程度の性能向上である．DhrystoneやWhetstoneでは2倍程度の性能向上であるが，メモリのスループットに関しては33％程度性能が低下する．マルチメディア命令のベンチマークでは性能に変化はなかった．まあ，Intelの公表程度の性能向上であるが，本当のところは正式なPrestonia版Xeonが登場するまではわからない．なお，IDF 2002 Springによると，Prestonia版Xeonはデュアルプロセッサ専用の廉価版「Xeon DP」として，動作周波数1.8G，2GHz，2.2GHzの3種類が登場するらしい．

一方，「本業」のPentium4(Foster)ベースの「Xeon MP」は2002年3月にドイツで開催されたCeBITで正式発表された．サーバ用ということでSMPサポートはもちろんであるが，512Kバイトまたは1MバイトのL3キャッシュとハイパースレッディング機能を内蔵する．動作周波数は，1.4，1.5，1.6GHzの3種類で，単体のPentium4(Northwood)が2.2GHzとなっているのに対して控えめである．性能はPentium IIIベースのXeon/900MHzに対して1.6GHz版で30％の向上らしい．実際のベンチマークでも同様の数値が出ている．フロントサイドバスの周波数も400MHzに向上し，動作周波数も78％向上しているのに，その程度の性能向上に留まるのは，Pentium4のIPCがPentium IIIに比べて低いためであると想像される．ただし，ソフトウェアもハイパースレッディングには未対応のようなので，将来的にはもっと性能向上するかもしれない．

表3　Intelプロセッサの分類

シングルプロセッサ	Xeon(DP，MP)	MPUのカテゴリ
Pentium	Pentium Pro	Pentium
Pentium II	Pentium II Xeon	Pentium II
Katmai	Tanner	Pentium III
Coppermine	Cascades	Pentium III
Tualatin	Pentium III-S	Pentium III
Willamette	Foster	Pentium 4
Northwood	Prestonia(DP) Gallatin(MP)	Pentium 4
Prescott	Nocona(DP) Potomac(MP)	Pentium 4 (Pentium 5)
Tejas	Jayhawk(DP) Tulsa(MP)	Pentium 4 (Pentium 5)

2002年11月14日，Intelはハイパースレッディングを(公式に)実装するPentium4/3.06GHzを発表した．ハイパースレッディングが利用可能なほかは，従来のPentium4のクロックアップ品とあまり変わらない．このPentium4-HTの発表と同時に，各Webサイトではベンチマークテストが積極的に行われた．その結果は，ハイパースレッディングをONにすると10%から18%程度の性能向上だった．当初の見込みよりはやや低めであるが，OS側の対応が完全でないせいもあるのだろう．また，定番の「SYSmark 2002」や「3Dmark 2001SE」では，ハイパースレッディングをONにしても，まったく性能向上しなかったという．ハイパースレッディングがまだまだこれからの技術であることを窺わせる結果だ．

当初，ハイパースレッディングとしてはPentium4の3.06GHz版のみを匂わせていたIntelであるが，2002年の11月中旬にはロードマップの変更を行っている．つまり，2003年の2Qには2.40/2.60/2.80/3.20 GHzのハイパースレッディング対応のPentium4を投入するとしている．これはAMDのAthlon64の発表を受けて，Pentium4シリーズの性能面のてこ入れを図ったという見方が大勢である．

5　マルチプロセッサ結合の実際

● Xeonの場合

IntelのPentium系プロセッサにはいろいろな種類があり，すぐには理解できない．基本的にはシングルプロセッサをSMP対応にすることでマルチプロセッサとしてサーバ向けに提供している．最近では，そのブランド名はおしなべてXeonとなっている．**表3**にPC用シングルプロセッサ(Hyper-Threadingを含む)とサーバ用デュアルプロセッサ(DP)とマルチプロセッサ(MP)との対比を示す．文献を読むときの参考にしてほしい．

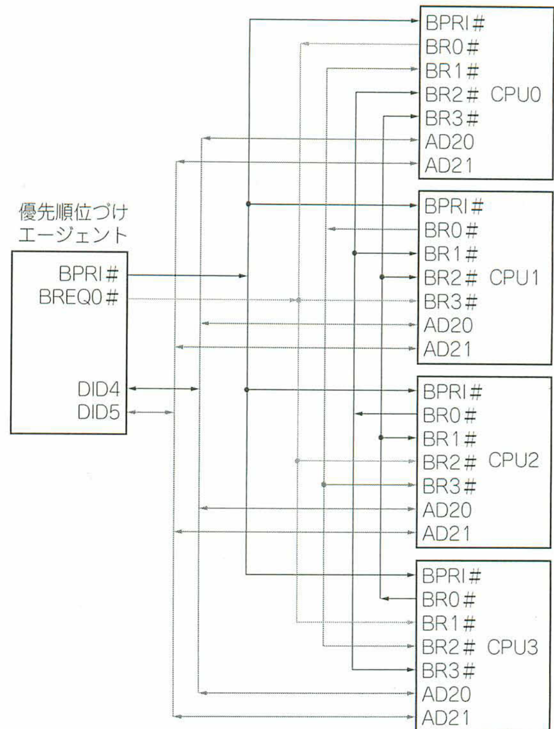


図15　各CPUの接続

なお，IntelではTulsaがデュアルコアになることを2003年秋のIDFで発表している．ただし，Hyper Threadingの実装は当然なので，ソフトウェアからは4CPUに見える．

▶ 接続方法

XeonのM版は，4個のCPUを**図15**のように，BR0#～BR3#(Bus Request)という4本の端子で相互に結合する．リセット時に，i840などのチップセットである優先順位づけエージェント(Priority Agent)がBREQ0#をアサートし，各CPUにCPU番号(Agent ID)を通知する．その後，BR0#は出力，BR1#～BR3#は入力端子として働き，これらがPriority Agentからのバス要求信号であるBPRI#とアービトレーションを行い，CPUはバスの使用権を獲得できる．

バスを使用したいCPUはBR0#をアサートして，その旨をほかのCPUに通知する(リセット時はBR0#をアサートされたCPUがバスの使用権をもっている)．このとき，そのCPUの優先順位が一番高い状態にあれば，バスの使用権を獲得できる．そして，BR3#がアサートされているCPUが次に高い優先順位をもつ．

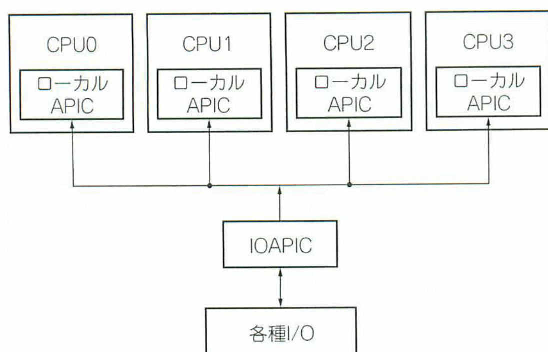


図16 APICの接続

直前にバスを獲得していたCPUはBR1#がアサートされることになり，優先順位がいちばん低くなる．このように，各CPUは公平(いわゆるラウンドロビン方式)にバスの使用権を獲得できるようになっている．また，Priority Agentからのバス要求は常に最高の優先順位をもっているので，BPRI#がアサートされた場合は，各CPUはただちにバスを明け渡す．

また，CPUはバスの使用権を獲得すると，AD20とAD21にAgent IDを出力する．これにより，現在バスの使用権をもっているCPUを知ることができる．

XeonのDP版では，BR2#とBR3#は(公式には)使用できない．そのため接続方法としては，二つのCPUでBR0#とBR1#をクロスして接続することになる．BR2#とBR3#自体は隠し機能としてチップには存在するようなので，DP版でも4CPU構成の接続にはできるらしい(もちろん，メーカーの保証外)．しかし，その場合でも，BIOSの変更(CPUの初期化など)が必要になるだけでなく，CPU自体が2や3というAgent IDに対応していないらしく，実際に動作させることは不可能なようである．

▶APICとハードウェア割り込み

IntelのSMP仕様では，割り込みによって動作するCPUを決定するのでAPIC(Advanced Programmable Interrupt Controller)の存在が欠かせない．チップセットに内蔵されたIOAPIC(I/O APIC)とCPUに内蔵されたローカルAPICでハードウェア割り込みを拡張して，各CPUに割り込みを割り振るようになっている．

Intelは1994年4月に，マルチプロセッサ向けのアーキテクチャとしてMPS(Multi Processor Specification)，およびそのための割り込みコントローラとしてAPIC(Advanced Programmable Interrupt Controller)を発表した．

APICは，初期の段階では，実際に割り込みを受け付ける部分(I/Oユニット)と，そこからの割り込みをCPU側で受け付ける部分(ローカルユニット)とに分けられていた．しかし，中期のPentium(P54C)以降では，ローカルユニットはローカルAPICとしてCPUに内蔵されている(**図16**)．これらが，ICC(Interrupt Controller Communication)バスを通じて結合される．APICは周辺機器から通知されるハードウェア割り込みを，それらが処理されるべきプロセッサに割り振る働きをする．

APICを使用すればSMP構成を容易に実現できるが，APICはIntelが特許を所有している技術であるため，昔は，他のCPUベンダー(VIA TechnologiesやAMDなど)は，これを利用できなかった．その代わり，VIAやAMDでは，オープンな技術であるOpen PIC標準(1995年発表)をサポートしていた．しかし，OpenPICにしたがったシステムを提供するメーカーはなく，互換MPUではマルチプロセッサは不可能というのが常識になっていた．ところが，現在はAPICが工業標準(ISA)として公開され，Intelの特許を侵害しない範囲内でなら互換MPUメーカーやチップセットメーカーが実装できるようになっている．IntelのMPUの競合品であるAthlonにおいても，初期モデルを除き，APICが実装されている．

● R4400MCの場合

R4400MCは，MIPS社の開発したR4400のマルチプロセッサ対応版のMPUである．他のR4400と比べて，キャッシュのインタベンション(スヌープ)要求，インバリデート要求，アップデート要求，コヒーレントリード/ライト要求をサポートしている．

図17に4個のR4400MCを用いたマルチプロセッサシステムの構成例を示す．各R4400MCはL2キャッシュと外部エージェント(制御用ASIC)でサブシステムを形成している．これが，システムバスを介してメインメモリを共有する．これは典型的なSMP構成である．

外部エージェントは，プロセッサからの要求をシステムバスのトランザクションとして転送し，システムバスからのトランザクションをプロセッサへの要求に変換する．外部エージェントがシステムバスを使用する場合は，まず，システムバスの使用権を獲得する必要がある．システムバスの使用権を獲得するまでの間，プロセッサからの要求は待機状態になる．

▶ロードおよびストア

プログラムのリード/ライト要求に対してL1キャッ

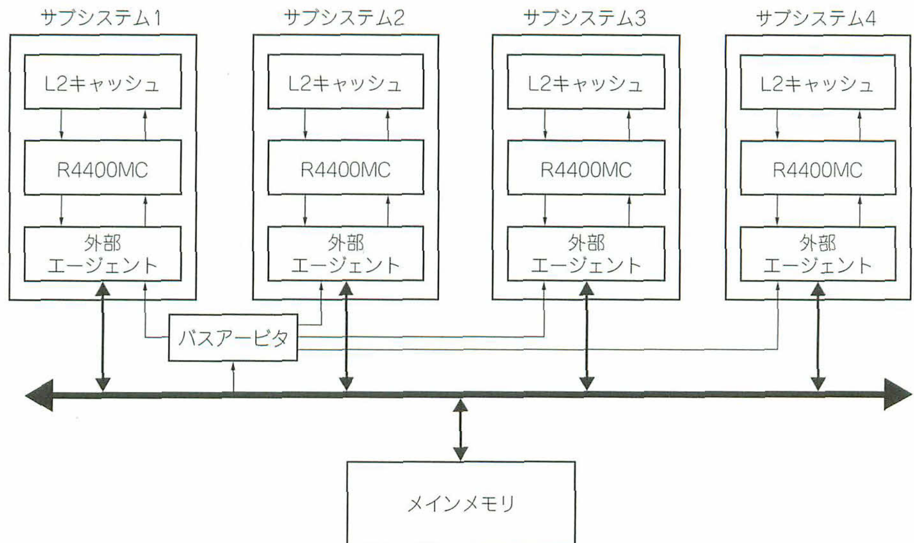


図17
R4400MCを4個使ったマルチプロセッサシステム構成例

シュとL2キャッシュがミスすると，R4400MCはコヒーレントリード要求を発行する．それを受けて，外部エージェントは，システムバスに対してコヒーレントリード要求を発行する．残りの三つの外部エージェントは，そのリード要求を受けて，自身が接続するR4400MCへの外部インタベンション要求に変換する．そして，その結果(対応するキャッシュラインの状態とデータ)を，要求元の外部エージェントに返す．そのリード応答の状態がSharedであるか否かによって，L2キャッシュに格納するデータの状態を決定する．リードミスに対しては，Shared状態であればそのまま，そうでなければClean Exclusive状態で格納する．ライトミスに対しては，MESIアルゴリズムを採用する場合はDirty Exclusive状態で，MOESIアルゴリズムを採用する場合はDirty Shared状態で格納する．

プログラムのライト要求が，Shared状態のラインにヒットする場合は，R4400MCはインバリデート要求(MESI)またはアップデート要求(MOESI)を発行する．外部エージェントは，それを受けて，他のサブシステムへのインバリデート/アップデート要求に変換して，システムバスに転送する．その要求に対する受け付け応答が返ってくるとき，ライト要求は完了し，キャッシュラインはDirty Exclusive状態(MESI)，またはDirty Shared状態(MOESI)となる．

▶ コヒーレントリードとリード応答

外部エージェントは常にシステムバスを監視している．しかし，システムバス上にコヒーレントリードを検知してもすぐには応答を返さない．まず，自身が接続するR4400MCにインタベンション要求を発行し，キャッシュ内に要求されているデータが存在するか否かを調べる．データが存在すればシステムバス上にSharedを通知する．また，対象となるキャッシュラインがDirty Exclusive状態であれば，そのキャッシュラインのオーナーになることを示すため，Takeover(引き継ぎ)を通知する．Takeoverを通知する場合は，Shared状態だけではなく，リード要求に対する応答データもシステムバスに出力する．要求元の外部エージェントは，その応答データを引き取って，R4400MCへ返す．同時に，応答データはメインメモリにライトバックされる．

なお，リード要求に対する応答データは，すべての外部エージェントが外部インタベンション要求を完了した後に返ってくる．

▶ インバリデートとアップデート

R4400MCが発行するインバリデート/アップデート要求に対し，外部エージェントは，そのままインバリデート/アップデート要求をシステムバスに出力する．

他の外部エージェントは，システムバス上にインバリデート要求を検出すると，対応するキャッシュラインを無効化する．システムバス上にアップデート要求を検出すると，それに付随するデータを引き取り，対応するキャッシュラインにShared状態で格納する．

▶ ライト

外部エージェントは，システムバス上のライトトランザクションに対しては，何も反応しない．ライトデータはメインメモリに格納されるのみである．

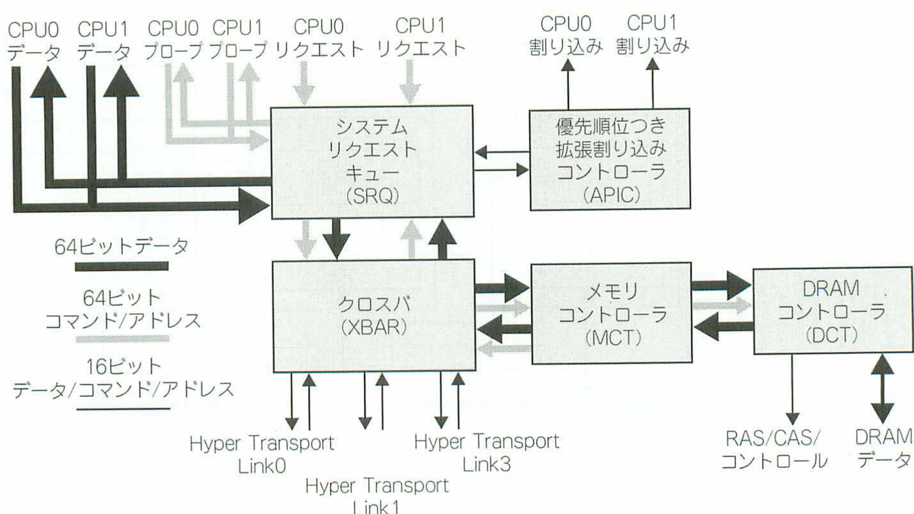


図18　Hammer内蔵のNorth Bridgeの構成

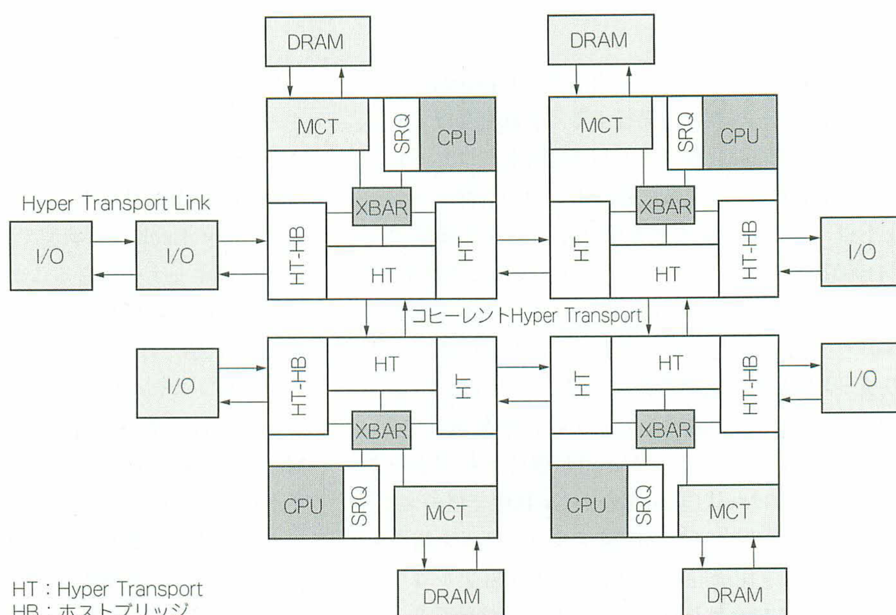


図19　Hammerにおけるマルチプロセッサ構成時の接続例

● Hammerの場合

Hammerに内蔵されるNorth Bridgeの構成を図18に示す．1チップ内に，プロセッサ，APIC，システム要求キュー(SRQ：System Request Queue)，DDR SDRAMインターフェース，3ポートのHyperTransportが集積されている．これにより，外付けのチップなしにマルチプロセッサ構成を可能にできる．

Hammerにおけるマルチプロセッサ構成時の接続を図19に示す．ここでは四つのチップを，2ポートのHyperTransportでリング上に双方向接続している(HyperTransportの残り1ポートはI/Oに使用)．プロセッサ間の通信はキャッシュコヒーレンシに対応したCoherent HyperTransportで行う．この構成は，基本的にはメモリの位置によってアクセス時間が異なるccNUMAである．しかし，AMDは全プロセッサのキャッシュのスヌープ結果を集めてコヒーレンス制御を行うアーキテクチャであり，実質SMPであると主張している．非公式資料によれば，AMDはHammerのメモリアーキテクチャをSUMO(相撲：Sufficiently Uniform Memory Organization)と呼んでいるらしい．NUMAは，スケーラビリティは高いがソフトでの管理が複雑であり，SMPはソフトは作りやすいがスケーラビリティが悪いということで，

(1) 単一のメモリ空間でキャッシュコヒーレンシが維

持される

(2) ローカルとリモートメモリのアクセス時間の差がDRAMのページヒットとミスの差と同程度

を特徴としているらしい．現在のSMPではどこのメモリをアクセスしても完全に対称ということはないので，Hammerの方式はSMPと大差ないと思われる．

図20に，CPU1がCPU3のリモートメモリをリードする場合のトランザクションを示す．各チップのローカルメモリのコヒーレンシを保つために，要求を受けたCPU3は各CPUにProbe（スヌープ）要求を送り，それを受けたCPUは自身のProbe結果を要求元のCPU1に返す．この時点で，CPU1はどのCPUからデータを受け取るべきかを判断する．その後，CPU1に要求したメモリデータが返り，CPU1はデータを受け取ったという応答をデータの転送元（CPU3）に返す．

(1) CPU1が隣のCPU2に読み出し要求（Read Cache Line）を送る．

(2) CPU2がその要求をCPU3に中継する．

(3) CPU3は要求を受け取り，自身のキャッシュのProbe（スヌープ）とメモリのリードを行うと同時に，両隣のCPU0とCPU2にProbe Request（スヌープ要求）を送る．

(4) CPU0はProbe RequestをCPU1に中継する．CPU3は自身のProbe結果（Probe Response）をCPU2に送る．

(5) メモリからの応答（Read Response）がCPU3に返る．CPU0は自身のProbe結果をCPU1に送る．CPU2はCPU3のProbe結果をCPU1に送る．

(6) メモリからの応答がCPU2に中継される．CPU2のProbe結果がCPU1に返る．この時点で，リード要求を出したCPU1に全CPUのProbe結果が集まる．

(7) メモリからの応答がCPU1に返る．

(8) CPU1はメモリデータを受け取ったという応答（Source Done）をCPU0に送る．

(9) CPU0はSource Done応答をCPU3に中継する．

以上の動作は，CPU3の下のメモリに最新データがある場合，つまり，どのCPUのキャッシュもオーナになっていない場合の話である．オーナならば，DirtyまたはShared Dirty状態にある最新データをもっている．

CPU3からProbe結果を逆送していく過程で，どれかのCPUがオーナであることが判明すれば，そのCPUからCPU1へのProbe Responseに続けて最新データを送ることが可能である．この場合，CPU3に

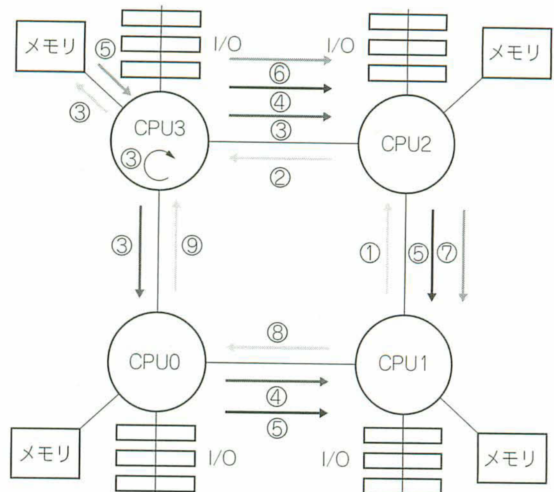


図20　CPU1がCPU3のリモートメモリをリードする場合のトランザクション例

はProbe Responseは返らないので，CPU3の下の（最新ではない）メモリデータもCPU1に返ってくる．CPU1は，全CPUのProbe結果を知っているので，古いデータは破棄すればよい．

これまでの説明は非常に単純な場合である．しかし，二つのCPUが同時にメモリリード要求を発行した場合のアービトレーションはどうなるのか，また同一のメモリに二つのCPUがライト要求を発行した場合に最新のデータがメモリに残るのか，などというコヒーレントアクセスの競合の問題がある．このような複雑な場合については不明であるが，明らかになったとしても，説明に文字数を費やすだけなので，ここでは言及しない．

● Power4の場合

▶ チップ間の接続

Power4システムでは，CMP構造のチップを4個搭載したMCMがシステム構成の基本になる．それを4個リング状に接続することで，16チップ，32CPUのマルチプロセッサシステムを構成できる．一つのMCMは8CPUのSMP構成で，MCM間はDistributed Directoryを使ったccNUMA構成だという．Power4はプロセッサ間の接続機能をチップに内蔵しており，外付けASICなしでマルチプロセッサシステムを構成できる．

各Power4チップには，2個のCPUとL2キャッシュ，L3キャッシュのディレクトリ（タグ）が集積されている．これにL3キャッシュとメモリを接続したものをリング状に4個並べて1枚のMCMが構成される．各チップはL2キャッシュとL3の中間に位置するFabric

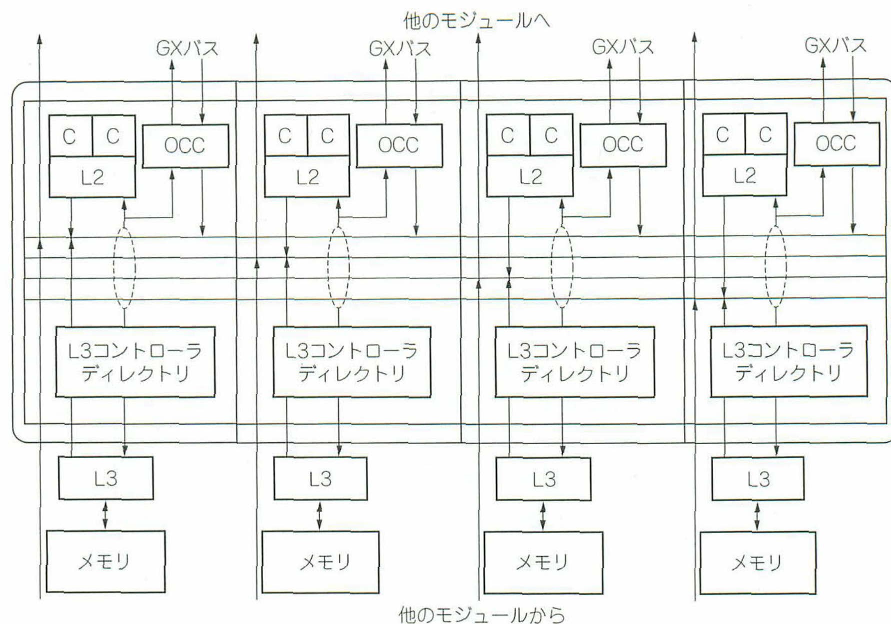


図21
Power4のMCM構成

Controllerで相互に接続される(**図21**). L2キャッシュは7状態, L3キャッシュは5状態のキャッシュコヒーレンシプロトコルを採用し，一般的なMOESIなどより賢い動きをする．これには，キャッシュ中のデータをできるだけ有効利用することでL3キャッシュへのアクセスやメインメモリへのアクセス回数を減らすことを目的するIBMの特許が適用されているらしい．

他のMCMとの接続はチップ単位に行われる．つまり，MCMがCHIP0, CHIP1, CHIP2, CHIP3から構成されているとすると，CHIP0どうしのリング，CHIP1どうしのリング，CHIP2どうしのリング，CHIPどうしのリングを作り，全体として4重のリングになる(**図22**)．これをイメージで示すと**図23**のような感じになる．

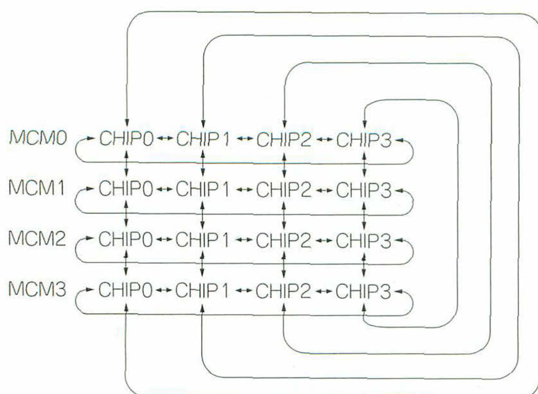


図23　Power4の32CPU構成時の接続イメージ

このように，32CPUという比較的大型のマルチプロセッサシステムをコンパクトに構成している．このため，リング構造を使っても，各チップ間の距離は短く，メインメモリも近くなり性能が向上する．各Power4チップのL3キャッシュのディレクトリはこれらのシステムのキャッシュコヒーレンシを維持するためのccNUMA管理機能も有していると推測される．

ところで，Power4のシステムは接続方式が厳密に規定されて設計されているように思えるので，このままでは32CPUを超えることはできないだろうと思われる．それ以上に拡張する場合は，GX Portを経由したccNUMAシステムとなると予想されるが，現実には32CPU以上のシステムは発表されていない．

▶ 実際のシステム構成

Power4を搭載した最高位のシステム(2002年時点)はIBMのeServer p690である．これは，当初の設計どおり，4枚のMCMを使用した32CPUのシステムである．しかし，HPCという科学技術計算用の構成もある．これはPower4チップ内の2CPUのうち片方を殺している．こうすることにより，通常は2CPUで共有しているL2キャッシュ，L3キャッシュ，メモリが1CPU専用になるので，CPUの処理能力よりもメモリ性能が重要とされる大規模なデータを扱うシステムで有利になる．IBMはそのような数値計算において，30～40％の性能向上が見込めるとしている．なかなか興味深い結果である．

p690の下位機種にp670がある．これはPower4の4

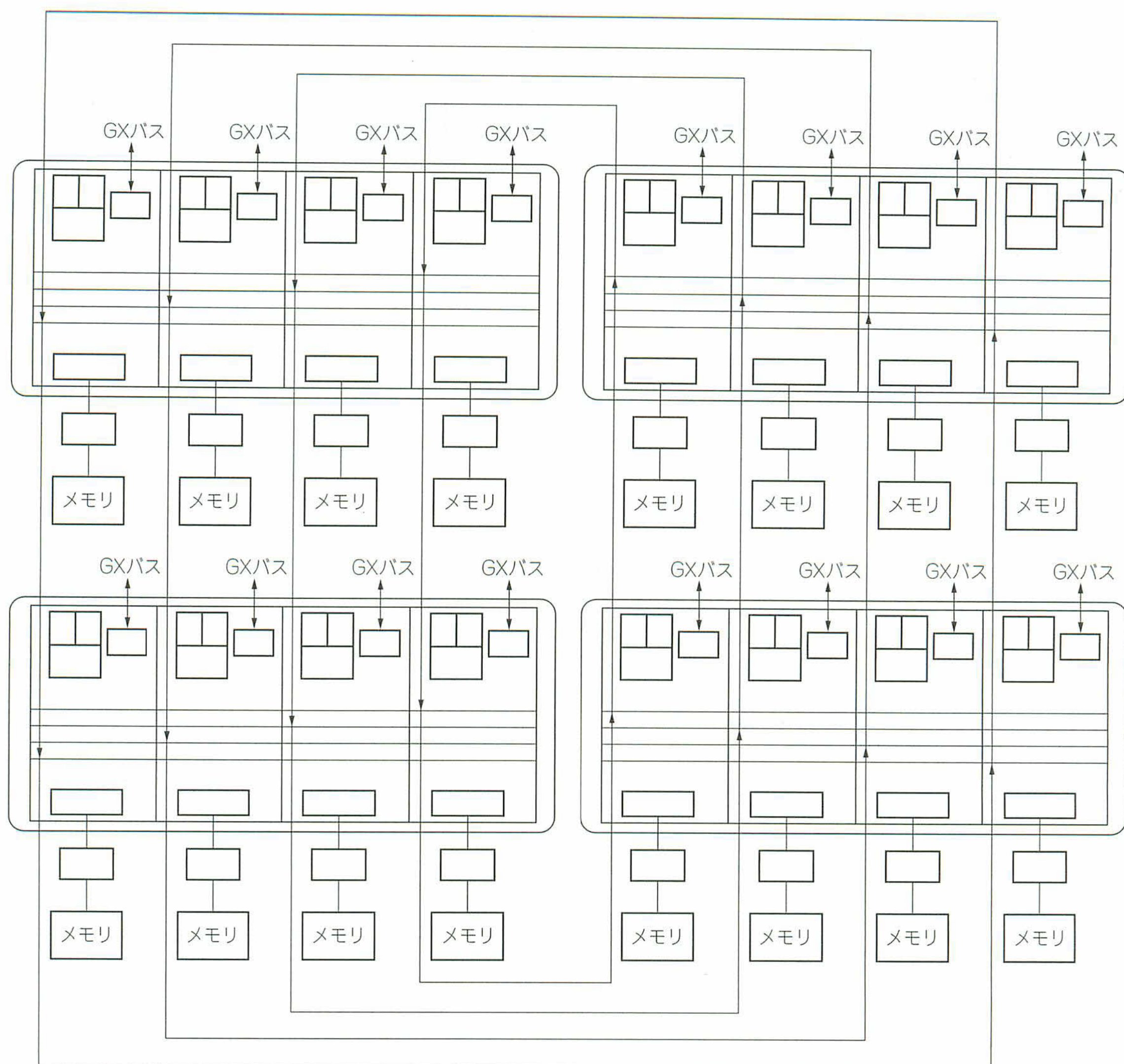


図22　Power4の32CPU構成

～16CPU，つまり最大2MCMのシステムである．このうちの4CPUシステムは興味深い．1枚のMCMで構成され，Power4も4チップ搭載されているという．これはHPCモデルのように2CPUの片方を殺しているのであろうか，それとも，片方のCPUが不良のPower4を利用しているのだろうか．8CPUと4CPUでは，最小構成において，約9万ドルの差がある．

p670のさらに下位機種として，p630というのもある．最大4CPU搭載であり，1CPUの場合，Power4の片方のCPUは殺してあるという．上位機種とは異なり，Power4を1個のみ搭載するモジュール（MCMではない），L3キャッシュ，メインメモリを搭載したプロセッサカードを最大2枚搭載する．2枚のプロセッサカードは，通常はMCM内でチップを接続するために使用されるFabric Busで接続されているという．

まとめ

マルチプロセッサ構成の基礎を説明した．今回はコンピュータアーキテクチャというよりもOSのアーキテクチャの説明のほうが多かった気もするが，最後まで読んでいただけたであろうか．いろいろな話題が詰め込まれて乱雑な気がしないでもない．初めから広く浅くの方針である．ところで一応，R4400MC，Hammer，Power4の構成例を紹介したが，具体的なマルチプロセッサの構成方法はMPUごとに異なるし，マルチプロセッサ構成をサポートしていないMPUもあるので，詳細を知りたい方は，各MPUのマニュアルを参照してほしい．

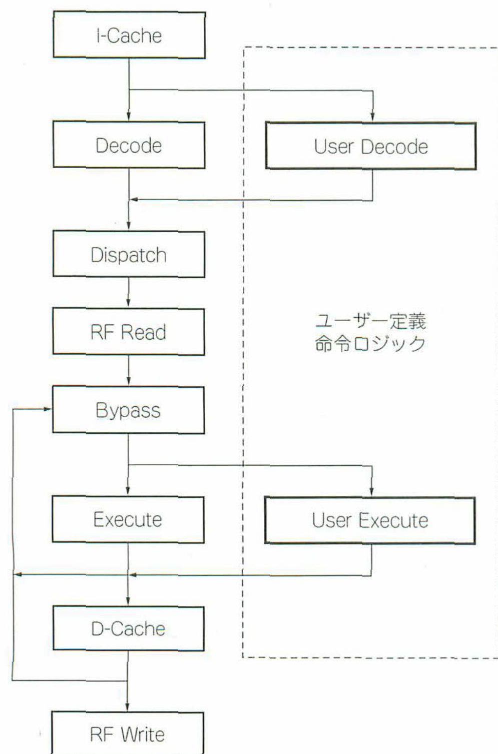
Appendix 5　コンフィギュアラブルの手法と実プロセッサの構成

コンフィギュアラブルプロセッサの概略

1 コンフィギュアラブルプロセッサの分類

●「柔らかい」回路技術

2002年頃から，プロセッサのアーキテクチャに関し，コンフィギュアラブル（configurable：構成可能）という単語が使われるようになった．これは，これまで御仕着せのハードウェアを利用するしかなかったソフトウェア技術者が，独自の設計で特定の専用機能を実現しようとするものである．あるいは，専用回路と同等の性能を実現しつつ，ソフトウェア並みの柔軟性をねらうものである．ソフトウェアを開発するのと同じ感覚でハードウェア開発ができるため「柔らかい」回路技術と呼ばれている．



図A　MIPSアーキテクチャのUDI理論

コンフィギュアラブルな機能を実現するプロセッサをコンフィギュアラブル・プロセッサと呼ぶが，そのコンフィギュアラブルにはいくつかの段階がある．たとえば，大ざっぱには次の三段階に分類できる．

- **第一段階**：ユーザーが独自の命令を追加できる．
- **第二段階**：プロセッサの設計時に回路要素を自由に構成できる．
- **第三段階**：プロセッサの動作時に回路要素を自由に構成できる．

第一段階は，一般的なIPコアでも導入されるようになってきている．たとえば，ARM社と並ぶIPコア提供会社の老舗であるMIPS社は，MIPSアーキテクチャのリリース2においてUDI（User Defined Instruction）というユーザーが自由に命令を定義できる機能を提供している．具体的には，命令コードの16個をユーザー用に開放してあり，その命令コードのデコード方法，使用する演算器を自由に定義できる．これにより，特定の応用分野において速度を要求される機能を1命令で実現できる（**図A**）[注1]．

ただARM社はユーザーのオリジナル命令の追加に否定的である．特定のデバイスに特化しても意味がなく，特定命令はユーザーの使用法を限定してしまい，結局は互換性がなくなるという主張である．

第二段階は，あらかじめ用意されているいくつかの回路要素をユーザーが自由に組み合わせて，ユーザー独自のカスタムチップを作ることである．これは従来からあるFPGA（Field Programmable Gate Array）の手法に似ている．FPGAに関しては後述する．

この第二段階の機能を提供するプロセッサ例としては，Tensilica社のXtensaを挙げることができる．

注1：MIPS社はUDI機能をCorExtendと命名し，その機能を提供するIPコアをPro Seriesと呼んでいる．2003年1月時点で，Pro Seriesには，4KE Pro（Emerald）ファミリ，M4K（Quartz）および4KSd Pro コアがある．2003年6月には24K（Topaz）がそのラインアップに加わった．

Xtensaはユーザーが独自なハードウェアを構成できるだけではなく，第一段階でいうところの新規命令も定義することが可能であり，FPGAとは一線を画している．

第三段階は，回路要素の組み合わせをプロセッサの動作時にも実行できる機能である．つまり，一つのプロセッサが，ある瞬間ではとある特定の専用機能，またある瞬間では別の特定機能を実現できるようになる．これは，MPEGのエンコーダとして最初は動作していたプロセッサが，後にはMPEGのデコーダとして動作するようなものである．このような機能は，コンフィギュアラブルといっても，再配置(リコンフィギュア：re-configure)可能という意味からとくにリコンフィギュアラブル型と呼ばれる．

第三段階の例としては，アイピーフレックス社のDAP/DNAやNECエレクトロニクスのDRPがある．

● コンフィギュアラブルはFPGAの進化形なのか

コンフィギュアラブルプロセッサは，数種類の回路要素を数百個程度1チップに内蔵してあり，それら回路要素間の配線をセレクタなどで切り替えて目的の回路を作り出す．この考え方はFPGAの考え方と同じであり，FPGAの特許に抵触するのではないかという懸念もあるそうだ．しかし，コンフィギュアラブルプロセッサの開発者は，その使い方がFPGAと一線を画しているとして，特許には抵触しないとみている．それどころか，コンフィギュアラブルプロセッサの基本技術を新規技術として積極的に特許を取得している，Tensilica社のような会社もある．

筆者は，静的に回路を構成する(上述の第二段階にあたる)コンフィギュアラブル技術は，特許抵触という観点からは灰色に近いと思う．しかし，上述の第三段階にあたるリコンフィギュアラブル技術は，回路の再構成時間がFPGAよりも高速(数10ms→数ns)である点，実行時のアプリケーションプログラムで再構成を指定できる点が新規技術であるといわれている．

ここで，FPGAに関して簡単に説明しておく．

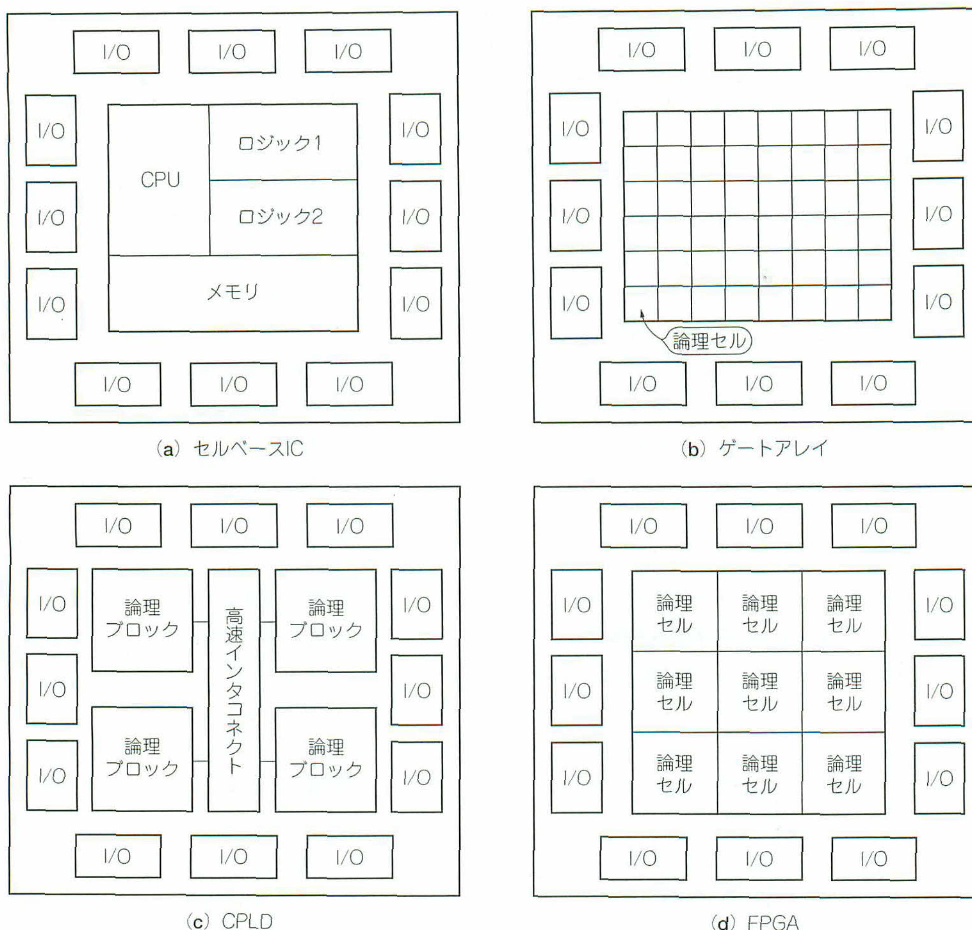
プロセッサの製造方式には，半導体製造メーカー寄りかユーザー寄りかでいくつかの段階がある．その昔(というか今でも)，ユーザーがある機能をもったハードウェアを欲しい場合は半導体製造メーカーに製造を委託して製造していた．これがカスタムチップあるいはASCP(Application Specified Custom Processor)と呼ばれるものである．

カスタムチップのうち，もっともチップ面積が小さくて高速動作を行うことができるのがセルベースICである．これらのICは，ユーザーの要求にしたがって基本セル(単一ゲートや複合ゲート)を配置・配線して製造する．あるいは，特定機能を実現するためのマクロセルとしてIPコアを組み合わせて製造する場合もある．これは，機能設計やタイミング設計などの点で製造が難しいので半導体メーカーに一任される．このため，チップ面積は小さくなり，速度が向上するなど，回路の最適化が可能になる．

ユーザーが機能設計を行って，半導体メーカーが製造するのがゲートアレイ(Gate Array)である．ゲートアレイは標準的な回路要素が1チップ内に論理セルとして多数内蔵されている．その配線方式をチップの製造時に指定することで，目的のハードウェアを実現する．ユーザーが設計したHDLの論理を論理セルの配線方式に変換する作業をマッピングという．ゲートアレイも高速動作が期待できるが，未使用の(むだな)論理セルが存在するため，チップ面積は大きくなる傾向にある．ただし，大枠の回路があらかじめ決まっているので，セルベースICと比べると安価に設計できる．また，配線工程だけでチップが製造できるので工期が短い．セルベースICやゲートアレイはASIC(Application Specified Integrated Circuit＝特定用途に特化したIC)の一般的な手法である．

FPGAは，ゲートアレイをユーザーが自分で設計できるようにしたものと考えれば理解しやすい．つまり，設計したい論理回路を設計現場で即座に実装できる．このようなデバイスを総称してPLD(Programmable Logic Device)という．つまり，論理をプログラム可能なデバイスという意味である．このPLDを大規模にしたものがCPLD(Complex Programmable Logic Device)であり，さらに大規模にしたものがFPGAである．CPLDとFPGAの違いは，内蔵されている論理セルの配線方式にある．CPLDは積項(プロダクトターム)と呼ばれる回路で論理セル間を配線する．FPGAはLUT(Look Up Table)と呼ばれるメモリとマルチプレクサを組み合わせた回路で論理セル間を配線する．セルベースIC，ゲートアレイ，CPLD，FPGAを比較したイメージ図を**図B**に示す．

ユーザーがHDLで記述した機能をデバイスにマッピングして専用回路を作るという点で，CPLD/FPGAはゲートアレイと同じである．しかし，何度も配線論理を書き換えできる点でゲートアレイよりも優れている．しかし，半導体メーカーが作成するセルベース



図B セルベースICとゲートアレイ，CPLD，FPGAの違い

ICやゲートアレイよりは低速である．高速動作を必要としない分野ではCPLD/FPGAをそのまま専用プロセッサとして利用する場合もあるが，一般としては，ユーザーが設計した回路機能をゲートアレイ化する前の動作検証用にFPGAを使用する場合が多い．計算機の機能シミュレーションで，HDL記述を用いて動作検証を行う場合は非常に多くの時間を必要とする．しかしFPGAで動作させれば，機能シミュレーションの数万倍以上の速度で検証できるため，デバッグが容易である．また，そのデバッグの高速性のため，機能シミュレーションでは実現できないような複雑なタイミングも検証することができる．

というのが少し前までの感覚だったが，今は少々事情が異なる．FPGAの進化には目覚ましいものがある．ゲート規模は数百万ゲートを遥かに超え，動作周波数も200MHz/400MHzは当たり前である．また，FPGAの老舗であるXilinxなどは，FPGAに可能なIPコア(CPUを含む)を各種取り揃えており，それらをSoC的に組み合わせるだけで専用チップが設計できてしまう．莫大な数量を必要としないならば，ASICやゲートアレイに頼らなくてもFPGAだけで期待した性能を実現することができる．つまり，比較的安価に専用チップを製造できるようになったのである．これは半導体メーカーにとっては脅威であり，また一方，FabレスなIPコアの論理設計のみという商売を加速するものになるであろう．

半導体メーカーもFPGA対策(？)には躍起である．短納期ASICというものが開発されている．通常のゲートアレイの1/5～1/4程度の期間で，顧客のネットリストを受け取ってから出荷までを可能にしている．このような製品には，LSI LogicのRapidChip，NECエレクトロニクスのISSP，富士通のAccelArrayなどがある．これですら工期的にはFPGAには敵わないが，動作速度，規模，消費電力の点でFPGAでは実現不可能な領域が確実に存在するというのが半導体メーカー(富士通など)の言い分である．

なお，コンフィギュアラブルプロセッサは，FPGAとASIC(ゲートアレイ)の中間的な用途で利用することを目的としている．

● 汎用プロセッサではダメなのか

何も無理してコンフィギュアラブルな専用プロセッサを利用しなくても，PentiumやAthlonのような汎用プロセッサを利用すれば，アプリケーションプログラムを切り替えることで，異なる種々の機能を実現できる．この点を考慮すれば汎用プロセッサのほうが融通が利く．その意味からすると，汎用プロセッサは本来コンフィギュアラブルなものである．世間で言うコンフィギュアラブルプロセッサは，プロセッサ内のデータ処理の構造までもコンフィギュアラブルにしたものである．

専用プロセッサが有利になる条件は何だろうか．特定用途の高速化であることは間違いないが，汎用プロセッサを複数個並列処理することで同等の性能を実現することはできる．専用プロセッサが選択されるためには，汎用プロセッサで同等機能を実現するよりも，システム的に低コストである必要がある．PentiumやAthlonはPC市場が活発なため「殿様商売」的なところがあり，あまりディスカウントをしていない．しかし，ARMやMIPSといった組み込み制御用のプロセッサは，通常なら1000円程度，高くても5000円程度である．これはかなり安価である．これらのプロセッサを複数個使用した場合よりも，コンフィギュアブルプロセッサを使用した場合のほうがシステムコストが安くなるという点が採用の分岐点である．現在，アイピーフレックスのDAP/DNA-HPは数千円程度といわれており，汎用プロセッサの5倍程度の性能差がなければ割に合わない．これはかなり微妙な分岐点であろう．

コンフィギュアラブルプロセッサの特徴は，主人であるメインのプロセッサと種々の専用機能を提供する家来のコプロセッサという構成である点だ．これは，汎用プロセッサとDSPの組み合わせや，汎用プロセッサと少し機能の劣る(MMUがないなど)小規模な汎用プロセッサといったシステム構成と同じである．実際このような構成を採用するSoCも多数存在しており，コンフィギュアラブルプロセッサのライバル関係にある．このような現状を考えると，コンフィギュアラブルプロセッサの独擅場になる場面をなかなか想定しにくい．

2003年6月の「2003 Symposium on VLSI Circuits」の1セッションにおいて，リコンフィギュアラブルプロセッサの実用化をめぐる議論が行われた．実際に製品を開発している会社(NEC，アイピーフレックス，SONY)は，リコンフィギュアラブルプロセッサは汎用プロセッサとASIC，あるいはFPGAとASICの間を埋める存在として必須であると主張した．しかし，高性能汎用プロセッサを開発しているIBMやIntelは，リコンフィギュアラブルプロセッサの潜在的な性能は認めるものの，再構成の時間がかかる点と再構成可能な回路構成が冗長(大規模)になる点を挙げて，汎用プロセッサの敵ではないと反論した．これらの論争がリコンフィギュアラブルプロセッサの賛否の現状を端的に表していると思っていいであろう．

リコンフィギュアラブルプロセッサの懐疑派は通信関係の人に多い．通信処理では各種のプロトコルを使うパケットが混在して入力されるため，プロトコルごとに構成を変更するのでは処理が間に合わないと主張する．2003年時点のリコンフィギュアラブルプロセッサ論議は，宗教的な論争の域を出ていないと筆者には思われる．

● C言語設計ではダメなのか

最近，SystemCなどのC言語で回路を記述するのが流行である．ソフトウェア屋さんが回路設計をするという目的からは，リコンフィギュアラブルプロセッサを使用しなくても，C言語設計をするということですべての要求を満たしていると思う．リコンフィギュアラブルプロセッサは，C言語などの高級言語で書いた記述から機能ユニット間の接続情報を生成するのに対して，C言語設計は直接回路を生成する．回路の効率的には，おそらくC言語設計のほうが勝るだろう．これは，従来のセルベース設計とFPGAの開発と容易に対比できる．これらと，C言語設計やリコンフィギュアラブル技術の相違は，意義的には高級言語で機能設計をすると機能検証が容易なので，設計期間が短縮できるところにあるのだろう．

だが現在のところ，C言語設計には，変換してできた回路情報が意図したものであるかどうかを検査する形式検証ツールに実用的なものがなく，その採用に躊躇しているエンジニアが多い．これが解決されるには，まだ数年を要すると考えられる．もしかしたらリコンフィギュアラブル技術はそれまでの隙間を埋める技術で終わってしまうかもしれない．

ダイナミックリコンフィギュアラブル記述は，通常のリコンフィギュアラブル技術とは性格が異なるの

で，それなりの意義はあると思う．しかし，通常のリコンフィギュアラブル技術は，将来的にはC言語設計に吸収されていくのではないだろうかと思う．

● 記憶素子を使ったコンフィギュアラブル技術

メモリに使われる記憶素子を利用したコンフィギュアラブル技術の研究を，ロームと東北大学が共同で行っている．

メモリといっても磁気で状態を記憶する強誘電体キャパシタ[注2]を使用している．強誘電体キャパシタを2個使用することでAND，ORといった論理演算を実現できる(詳しい理論は省略)ため，これらを組み合わせれば演算回路を構成できる．強誘電体キャパシタで論理回路を構成する利点は，回路規模が小さくなり動作周波数も向上することである．強誘電体キャパシタは本来が記憶素子であるため，回路内の記憶論理であるフリップフロップ，レジスタ，ラッチの置き換えが容易である．単純なフリップフロップでも数個の論理ゲートが必要だが，それを強誘電体キャパシタに置き換えると数分の1程度の大きさになることが期待できる(実際には1/3程度らしい)．また，回路規模が小さくなるのでゲート1段ごとに記憶素子を挿入することが可能になったり，演算器自体に記憶機構をもたせたりすることができるため，ゲート遅延を最小に抑えられる．ロームと東北大学が2002年2月のISSCCで発表した論文によると，強誘電体キャパシタを使用した54ビット×54ビットの乗算器[注3]を，0.6 μmプロセスで製造し，それは250MHzで動作したことが報告されている．通常の0.6 μmプロセスCMOS回路は100MHz程度の動作である．

強誘電体キャパシタは記憶素子であるため，FPGAで使用されるLUT(Look Up Table)や，従来のコンフィギュアラブル技術で利用されている回路要素間をつなぐスイッチの役割を果たすことができる．また，演算回路の間に記憶回路を容易に混在させることが可能なので，自由度の高い論理回路も構成できる．そして，マイクロアーキテクチャの足枷となっているメモリアクセスのボトルネックを解消することも期待される．このように記憶素子の利用は，次世代のコンフィギュアラブル技術として注目されている．

2 コンフィギュアラブルプロセッサの実際—静的リコンフィギュアラブル

いわゆるコンフィギュアラブルプロセッサにはいろいろな実装方式があり，その特徴を一般論として述べるのは難しい．しかし，2002年以降「コンフィギュアラブル」という属性は非常にクローズアップされている．ここでは，各社が提案しているコンフィギュアラブルプロセッサの実際について見てみたい．応用例を見る限り，各社ともDSPからの置き換えを狙っている感がある．

● Tensilica社のXtensa

XtensaはTensilica社が開発したコンフィギュアラブルで拡張性のある論理合成可能なプロセッサコアである．組み込み制御分野でのSoCとする目的でアーキテクチャ設計されている．目的とするSoCが要求する機能を実装できるようにコンフィギュアラブルな構造を採用している(**図C**)．

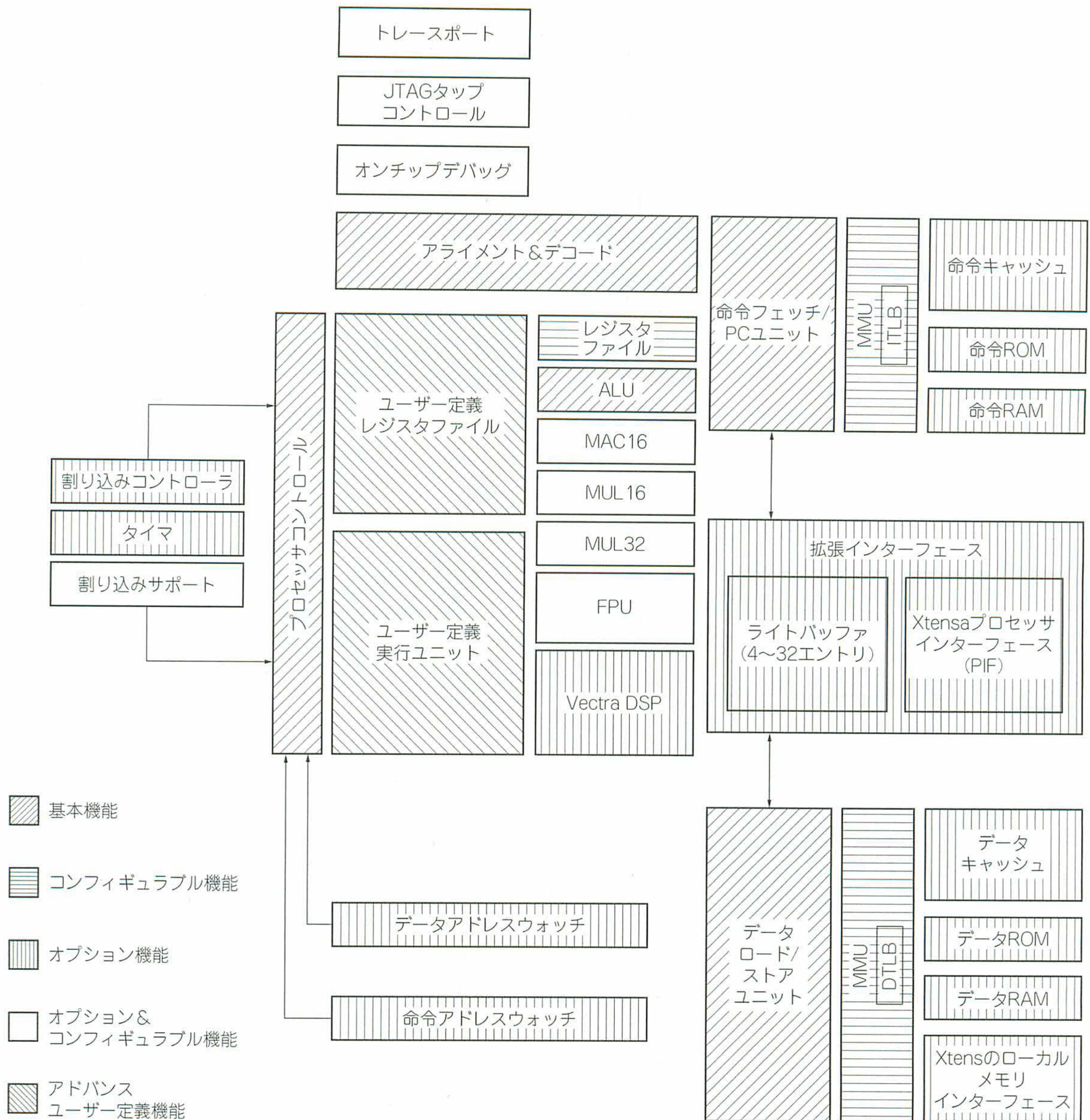
システム設計者は，あらかじめ定義されたアーキテクチャ要素を選択してコンフィギュア(構成)し，想定するシステムに最適な新しい命令や実行ユニットを作り出すことができる．これが従来のSoC設計とは大きく異なる部分である．

XtansaのRISCコアは組み込み制御向けのコンパクトなアーキテクチャをもっている．基本となるアーキテクチャは，1個の32ビットALUと最大64本の汎用レジスタ，6本の特殊用途レジスタ，80種の基本命令からなる．命令長は16ビットまたは24ビットであり，競合他社の32ビットに比べるとコードサイズを削減できる．命令セットで特徴的なのは，ループを最適化する比較-分岐命令，ファンネルシフトやフィールド抽出を含むビット操作命令があることだ．

システム設計者はTensilica Instruction Execution (TIE)言語を用いて，新規のデータ型，命令，実行ユニットを定義できる．TIEの記述をWebベースのXtensaプロセッサジェネレータに入力することによって，新しい論理合成可能なハードウェア情報を生成できる．また，Xtensaプロセッサジェネレータは RISCコアの構成要素を選択して高速にハードウェア

注2：強誘電体キャパシタでなくてもヒステリシス特性をもつ不揮発性メモリならば，理論的には何でも利用可能である．

注3：54ビットというビット数は倍精度浮動小数点演算器での応用をねらったものである．実際，乗算器の研究では54ビットの符号なし乗算を目的とするものが多い．



図C Xtensaのブロック図

を生成できる．選択可能な要素には次のものがある．

▶ **実行ユニットと命令セット**

- 乗算器：16ビットまたは32ビット
- Vectra DSPエンジン(**図D**)
- 浮動小数点ユニット

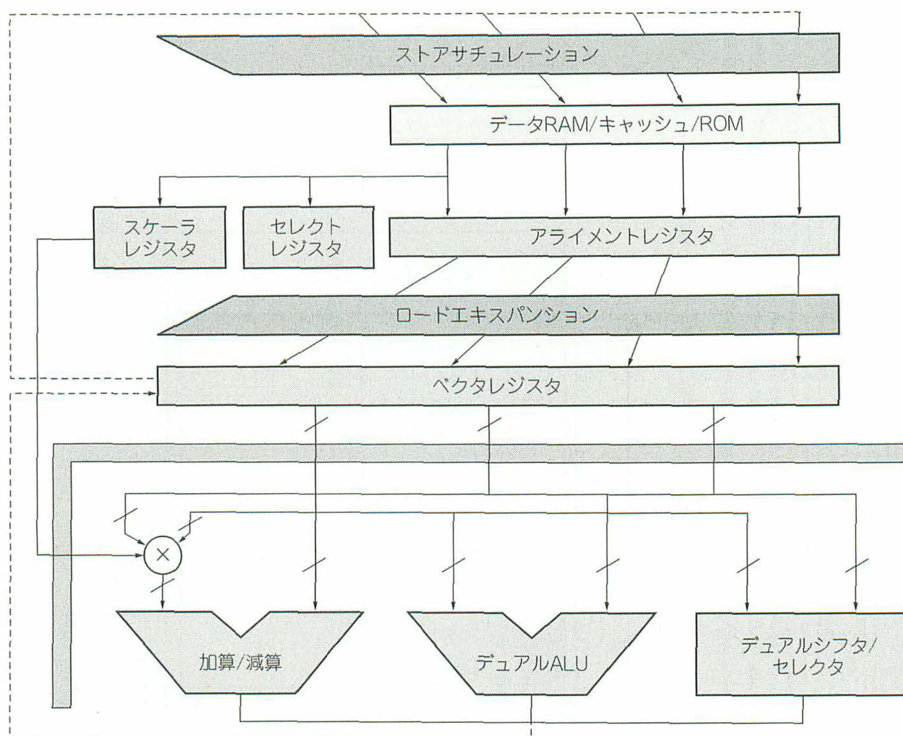
▶ **インターフェース**

- プロセッサインターフェース：32ビット，64ビット，128ビット
- ビッグエンディアン/リトルエンディアン
- オンチップのデバッグ機能
- 実行トレース用のポート
- 最大32本の割り込み

▶ **メモリシステム**

- MMU
- キャッシュ：最大4ウェイ，最大32Kバイト，ライト制御の選択
- 命令とデータでのROM/RAM領域の分割

また，Tensilica社はXtensaを将来的にVLIWとすべく，64ビット命令長をもつFLIX(Flexible Length Instruction Xtensions)を発表している．従来のXtensa向けオブジェクトコードもそのまま動作可能だ．そのしくみは，オブジェクトコード中に64ビッ



図D
Vectra DSPエンジン

16ビット	24ビット	64ビット(分割上位)	+24
64ビット(分割下位)		24ビット	+16
64ビット			+8
24ビット	16ビット	24ビット	+0

7	6	5	4	3	2	1	0	アドレス

図E　Xtensaのメモリ内の命令配置例(リトルエンディアン時)

ト長のVLIW型命令と，16ビット/24ビット長の従来のRISC型命令を，それぞれ独立に混在させることが可能にさせる(**図E**)．つまり，プログラム中で必要な部分のみをVLIW型命令で記述することができるのだ．ユーザーにしてみると，従来の命令に加えて64ビット長の命令が追加されたように見えるだけである．

● ARCのARCtangent

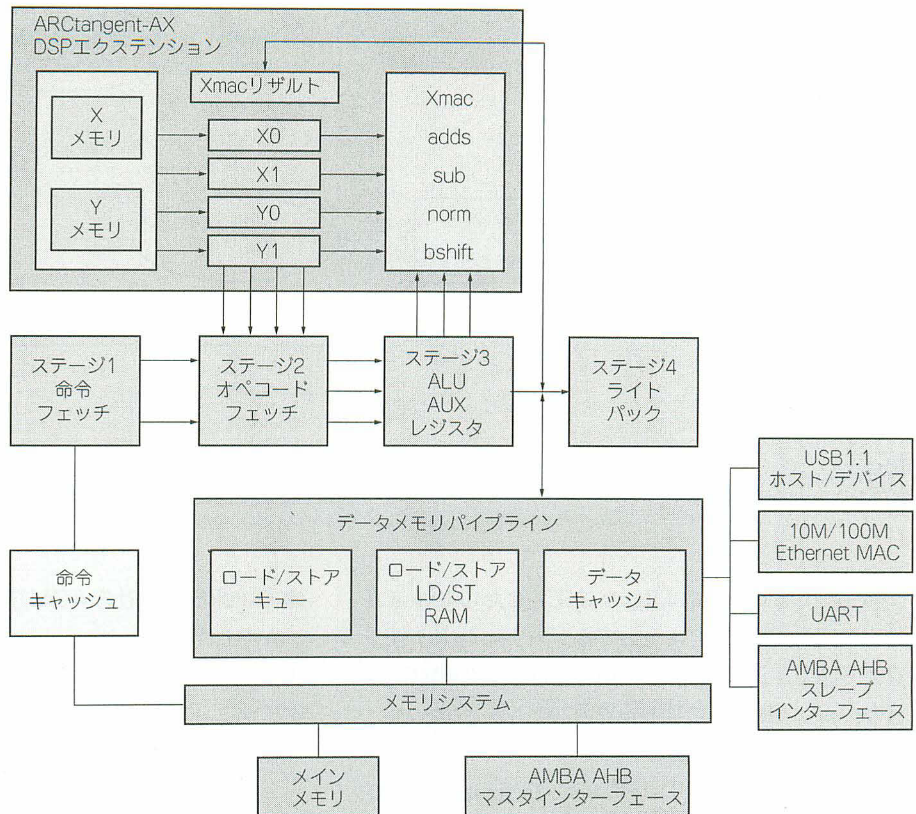
ARC International社(通称ARC Cores社)は，ネットワーク，通信，家電，無線分野のための論理合成可能なIPを提供するIPベンダである．1998年にゲーム開発会社のArgonaut Technology社から独立し，ほとんどすべての組み込み分野にカスタマイズ可能なコンフィギュアラブルプロセッサであるARCを開発した．Argonaut社はSuper Nintendo向けのグラフィックスアクセラレータとしてコンフィギュアラブルなコアを任天堂にライセンスしたことで有名である．

ARC社のWebページでは「コンフィギュアラブルなSoCプラットフォーム技術における世界的リーダー」とうたっている．実際，ARC社はコンフィギュアラブルプロセッサの先駆者である．IPコアとしてCPUも提供し，8ビットの汎用プロセッサから，16ビットの8086/80186，32ビットのARCtangentファミリまで幅広い応用分野に対応している．

中でもARCtangentファミリのプロセッサは，ユーザーが命令や構成を応用分野に特化してカスタマイズできるRISC型のコンフィギュアラブルプロセッサである．具体的にはARCtangent-A4とARCtangent-A5プロセッサがあり，これらにDSPを付加した構成も可能である．また周辺機能として，EthernetMAC，UART，USB(USB1.1ホスト/デバイス，USB2.0 On-The-Go)のIPを内蔵できる．これらは，ARChitectというGUIツールで簡単に構成できる．

ARCtangent-A4は高い周波数が要求される分野に最適である．ARCtangent-A5は，16ビット長を基調とした命令セットを提供し，低価格分野への適用に向いている．

ARCtangent-A4のCPUコアは4ステージのパイプ



図F　ARCtangent-A4のブロック図

ラインを有する32ビットRISCである．その命令セットは直交しており，アドレッシングモードと演算機能の自由な組み合わせが可能である．また，ほとんどの命令において条件コードを用いた演算が可能である．**図F**にARCtangent-A4のブロック図を示す．

ARCtangent-A4では，命令セット，レジスタファイル，条件コード，キャッシュ，バスなどのアーキテクチャ的な特徴をユーザー自身が構成/拡張できる．

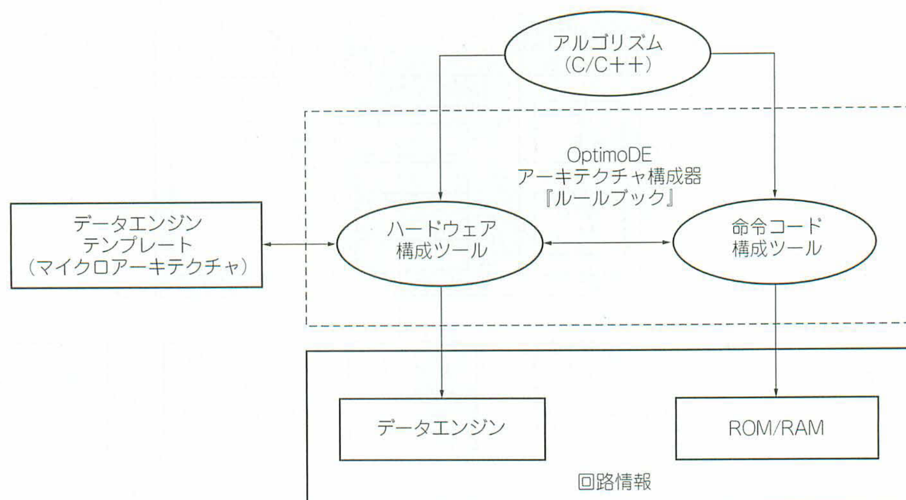
その命令セットは，標準的な命令が30個あり，さらに最高70種(2オペランドが16個，1オペランドが54個)の命令を追加できる．レジスタは基本が36本(うち汎用は29本)で，64本まで拡張できる．動作周波数は0.18 μmプロセスで200MHz，0.13 μmプロセスで260MHzである．また，プロセッサの面積は標準的な構成において，0.18 μmプロセスで0.18mm^2，0.13 μmプロセスで0.1mm^{2}である．

ARCtangent-A5は，基本的にはARCtangent-A4と同じものであるが，命令セットが異なる．命令セットには86個(35個が16ビット長，53個が32ビット長)の基本命令があり，応用分野に特化すれば128個の32ビット長命令と128個の16ビット長命令を追加できる．16ビット長命令を追加すれば，プログラムサイズを縮小することができる．レジスタは35本(うち26本が汎用)であり，さらに28本のレジスタを追加できる．動作周波数は0.18 μmプロセスで170MHz，0.13 μmプロセスで230MHzである．また，プロセッサの面積は標準的な構成において，0.18 μmプロセスで0.25mm^2，0.13 μmプロセスで0.13mm^{2}である．命令セットが複雑になった分だけ，動作周波数が低下し，面積も大きくなっている．

ARCtangent-A4とARCtangent-A5は，ユーザーが目的とする応用分野に応じて使い分けることができる．

ARC社は2000年7月にXilinxと提携して，自社のコンフィギュアラブルコアをFPGAとして提供できるようにした．これにより，ARCプロセッサとFPGAのどちらにも備えられたリコンフィギュレーションという特徴が，一般のプロセッサによるソリューションでは実現できない最適な柔軟性を提供できるようになった．

また，2003年10月，ARC社はARCtangent-A5のスピードと電力を改善したARC600を発表した．スピードを向上させるためにパイプライン段数を従来の4段



図G
ARMのOptimoDE概念

から5段に増やし，290MHzを達成させた(0.13 μmプロセス時)．消費電力についてもゲーティッドクロックを多用し，ARCtangent-A5が62mW@230MHzだったのが，ARC600では11mW@290MHzとなった．

● ARMのOptimoDE

OptimoDE(Optimal Data Engine technology)とは，ARM社が提供するVLIW構造のコンフィギュアラブルなコプロセッサである．とくにDSPと同様な制御を行うアクセラレータを自動生成する機能をもつ．

通常のDSPソリューションはアプリケーションのアルゴリズムを汎用プロセッサ(DSP)に合わせて開発する必要がある．しかし，OptimoDEでは，マクロアーキテクチャテンプレートに従って，アルゴリズム中の，ハードウェアで行う重い部分とソフトウェアで行う軽い部分を専用ツールが切り分ける．ハードウェアで実行する部分がVLIWのデータエンジンとなる．これは，システム構成に合わせてアルゴリズムを開発していた従来手法とは異なり，まずアルゴリズムありきで，それに適合したハードウェアを自動生成する手法である(**図G**)．

ARM社の発表によると，MPEG-2での応用例を挙げ，100%ソフトウェアで行うと611MHzの能力が必要な処理を240MHz程度で行うことができるという．この場合のアクセラレータの面積は1mm^2(プロセス不明，おそらく0.13 μm)であるという．

理論的にこの技術は，ARMに特化せずともほかのアーキテクチャにも流用可能である．ARM社が他社にライセンスするつもりがあるか否かは不明である．

● QuickSliver社のACM技術

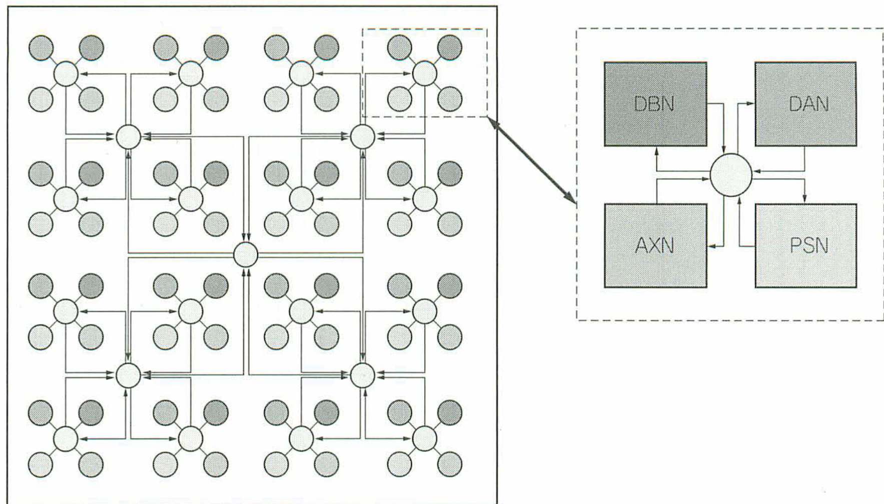
ACM(Adaptive Computing Machine)技術は，米国のベンチャー企業であるQuickSilver社が開発したダイナミックリコンフィギュアラブル技術である．

従来のリコンフィギュアラブル技術は，アプリケーションやアルゴリズムのレベルという大きな単位で考えられている．それらの多くは，ワード単位の処理は可能だがビット単位の処理は得意ではないということだ．つまり，ある構成から別の構成に変更するのが容易ではない．ACM技術は，このような問題を解決するために，局所的に専用機能を柔軟に組み合わせることを特徴としている．

この技術を一言で説明すると，スケーラブルで均一(homegeneous)なフラクタルネットワーク(Hilbert曲線か？)で，異種(heterogenious)の演算ユニットを接続して回路を構成する．つまり，4個の異なるノードを相互結合したノードをフラクタル状に結合するということである．代理店の説明によると，単純なプロセッサ(ノード)をCMP(Chip Multi-Processor)結合したイメージだそうだ．

概念は難しそうだが，基本的には4個の異種ユニットをI/Oで接続するレベル1クラスタをフラクタル(細密充填)上に接続することで回路を形成する(**図H**)ということになる．レベル1クラスタを形成する演算ユニット(Nodeと呼ぶ)には，

AXN … 複雑なALU
AN2 … 単純なALU
DAN … DSP算術演算
DBN … ビット操作



図H　QuickSilverのACM技術

DFN … 有限ステートマシーン
(Finite Sate Machine)
PSN … Scalar演算ユニット
XMC … 外部メモリコントローラ

など，全部で9種類ある．基本的には，

1) DSPライクプロセッサ
2) 32ビットRISCプロセッサ
3) ASIC並みのアクセラレータ

に分類される．どのようにネットワーク接続を変更するかについてはまだ謎が多い．

QuickSilver社は，低消費電力の携帯機器やワイヤレス通信機器に搭載するLSIにACM技術を利用することを考えている．実例として，音声コーデックのQCELP(Qualcomm Code Excited Linear Prediction)アルゴリズムの実装例を示している．DSPで実装した場合は消費電力が84mW，シリコン面積が4mm^2，ASICでは消費電力が18.45mW，シリコン面積が19.8mm^{2}になるという．一方，ACM技術を利用したPLDとRISCコアを集積したLSIでは，消費電力が10.9mW(PLDコアが2.9mW，RISCコアが8mW)，シリコン面積が7.15mm^2(PLDコアが5.15mm^2，RISCコアが2mm^2)で済むという．まあ，それなりの効果はあげているようである．

これをどのように販売していくかについては，AOIテクノロジーという会社がオリンパスや日商岩井の後ろ盾を得て計画している．AOIテクノロジーは嶋正利氏(あのインテルの4004を設計した)が，2003年10月から，社長に就任して注目を集めている．嶋氏はVMテクノロジーなどの社長を経験したが成功には至ってない．今回，ACM技術に惚れ込み，もう一度新しい技術でMPU設計をするという野望に燃えていると聞く．

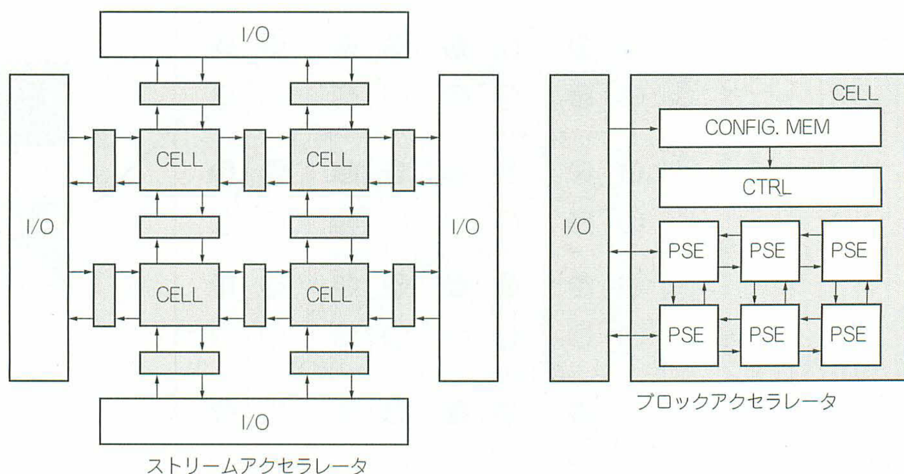
なお，2004年1月にACMの名称は「Adapt2000 ACM」と変更された．同時にテストチップのQS2412がリリースされた．プレスリリースによると，サポートされるノードはAXN，DBN，XMC，PSNの4種に限定されている．QS2412は12個のノードを搭載し，最大300MHzで動作する．消費電力は650mWである．

● Silicon Hive社のAVISPAとBRESCA

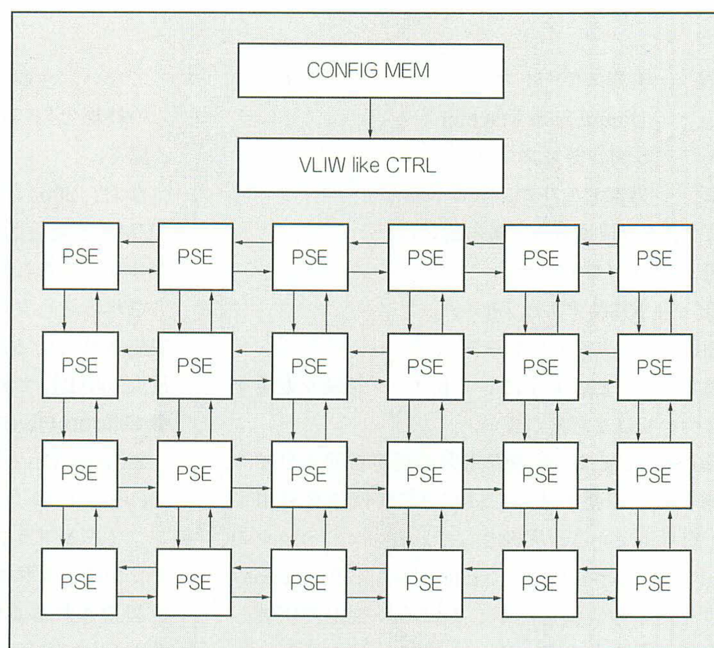
Silicon Hive社はPhilips(Philips' Technology Incubator)の子会社であり，Philipsの開発したリコンフィギュアラブル技術を応用した製品を開発・提供している．具体的には，C言語で記述された設計を基に製造される高性能で低消費電力のIPコアである．そのIPコアは，基本的にはDSPの置き換えをねらっており，各種汎用CPUコアのコプロセッサとして動作する．

Silicon Hive社の製品にはブロックアクセラレータと呼ばれるAVISPAと，ストリームアクセラレータと呼ばれるBRESCAがある．この二つは同一アーキテクチャであり，セル(CELL)と呼ばれるULIW(Ultra Long Instruction Word)で並列に制御する機能ブロックが一つのものがAVISPAで，それを格子状に相互結合して，さらなる性能向上をねらったのがBRESCAである(**図I**)．

一つのセルは複数のPSE(Processing and Storage Element)から構成されており，さらに各PSEは複数の発行スロット(Issue Slot)から構成される．これが



図I ブロックアクセラレータとストリームアクセラレータ



図J セルの構成例

ULIWの命令と1対1に対応する．セルとPSEの詳細を以下に述べる．

▶セル

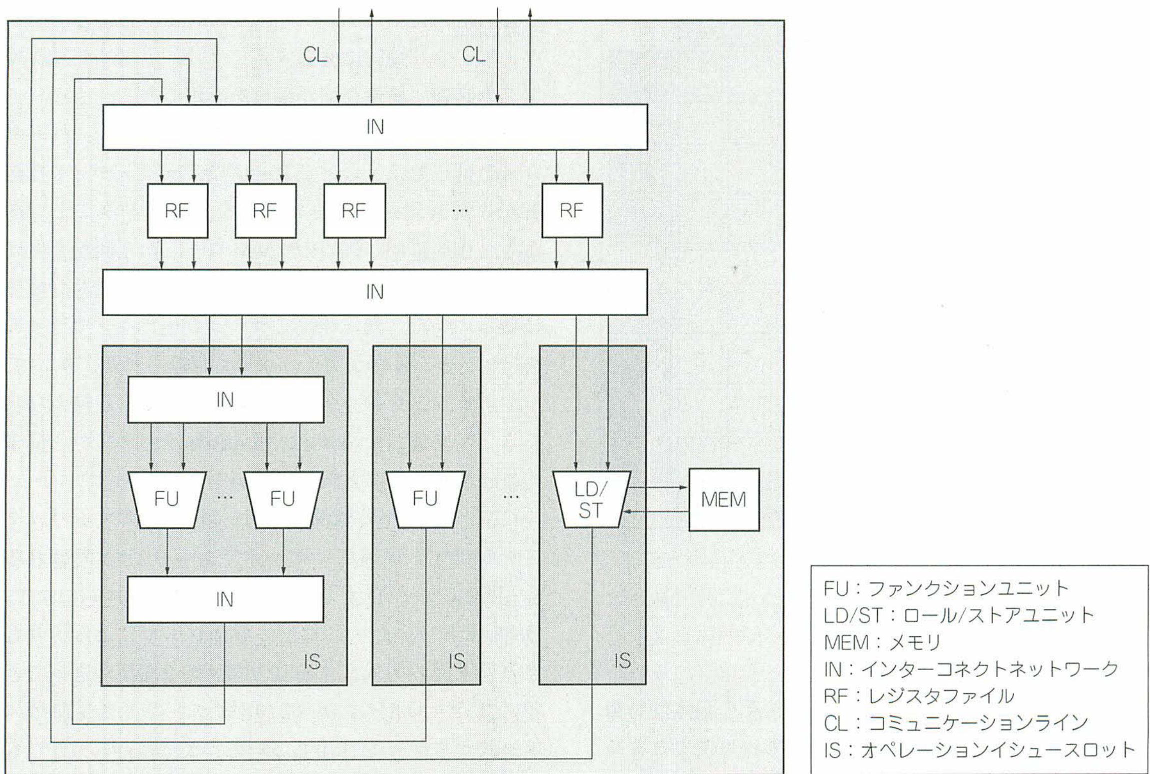
セルは，一つのVLIW似のコントローラ，一つのコンフィギュレーションメモリ，一つ以上のPSEのマトリックスで構成される(**図J**)．一つのセルが特定のアルゴリズムを計算するプロセッサとして成り立つ．セルの中のPSEは，CL(Communication Line)と呼ばれる信号線によってお互いに通信する．通常，一つのアプリケーションの機能は，一つのセルにマッピングされる．

▶PSE

PSEはアーキテクチャの基本的な構成要素である．PSEは複数の相互結合網(IN)から構成されるVLIW似のデータパスをもち，各機能ユニット(FU)と関連する一つ以上の発行スロット(IS)と結合される(**図K**)．機能ユニットは，オプションとして記憶素子(MEM)をもつ．

▶AVISPA

AVISPAの概略は次のとおり．



図K　PSE(Processing and Storage Element)**の構成**

- PSE×4，I/O PSE×1，サブプロセッサでシリコンを構成
- 最大512命令のULIW技術
- 41個の機能ユニットを並列実行可能
- 機能ユニットは75種類を備える
  (ビット操作，ALU，アキュムレータ，シフタ，ロード/ストアユニット，記憶素子)
- AMBAインターフェース

▶BRESCA

BRESCAの概略は次のとおり．

- 最大8×8個のセルで構成
- 一つのセルは14個の機能ユニットを含む
- 9個の機能ユニットを並列実行可能
- 32個の16ビットレジスタファイルを内蔵
- 16ビットの遅延メモリ
- AMBAインターフェース

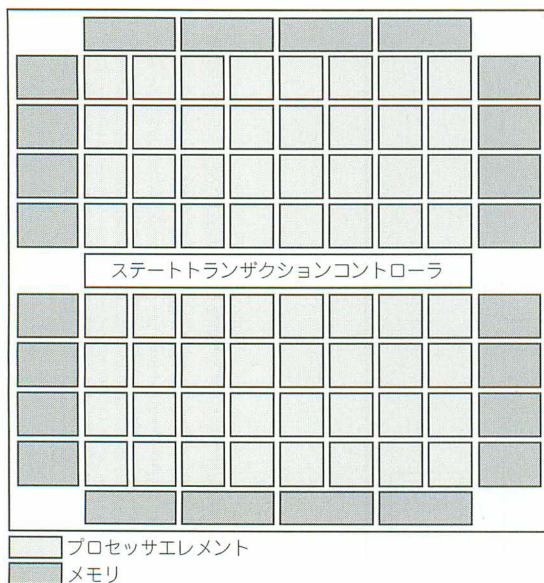
なお，2003年10月のMicroprocessor Forumでは，AVISPAを強化したAVISPA+の発表をしている．AVISPA+は，103個の機能ユニットと130個のレジスタファイルをもち，60個の命令を同時発行する．130nmプロセスでサイズは4mm^2，150MHz動作で9 GOPSの性能という．

3 コンフィギュアラブルプロセッサの実際 ―動的リコンフィギュアラブル

● NECエレクトロのクスのDRP

DRP(Dynamically Reconfigurable Processor)は，内部の構成を動的に変更可能ないわゆるリコンフィギュアラブル型プロセッサである．1チップ内に複数のPE(Processor Element)やメモリがアレイ上に配置されており，これらの要素の接続をクロックサイクルごとに切り替えることで種々の機能を実現する．具体的には**図L**に示すように，64個のPEと状態遷移を制御するSTC(State Transition Controller)と呼ばれるシーケンサを一つのタイルと呼び，このタイルを8個内蔵する．これは，ハードウェアによる高速処理をソフトウェア的に実現する技術であり，これにより，限られたチップ面積の中で仮想的に無限大のハードウェアを構成することが可能になる．

DRPは，並列プロセッサ上でネットワーク機器，プリンタ，車載機器といった専用論理回路と同等な処



図L　DRPのタイル構成

理を行うことを目指したものである．その最大の特徴は，動作中に内部構成を変更することが可能なことである．一つのプロセッサで複数の専用回路をもつプロセッサを作り出すことができるため，それぞれ個別に専用論理回路を組む場合に比べてチップ面積を削減できる．

DRPの評価チップは0.15 μmルールのCMOSプロセスで製造されている．幅広くシステムアプリケーションを評価できるように，動的構成部分は512個のPEと16面の動的回路構成面および2.2Mビットのメモリから構成されている．ただし，処理内容によってクリティカルパスの長さが異なるため，動作周波数は33MHz～133MHzの幅があるという．この動作周波数の幅がDRPの欠点ともいえる．つまり，想定する動作周波数によって構成できる専用回路の種類が制限されてしまうのだ．

NECエレクトロニクスによると，DRPは2003年以降の製品化を目指す．2004年には90nmのCMOSプロセスで250MHz動作品を発表する予定だという．

● アイピーフレックス社のDAP/DNA

DAP/DNAはアイピーフレックス社の開発したリコンフィギュアラブル型プロセッサである．同社独自のRISC型32ビットCPUコア「DAP(Data Application Processor)」と，148個のエレメント(演算器)がマトリックス状に並ぶ再構成可能な「DNA(Distributed Network Architecture)」の二つの部分からなる(図M)．DNA内部のエレメント間の配線をソフトウェアにより動的に変更することで，アプリケーションでの処理に最適な回路構成を実現できる．

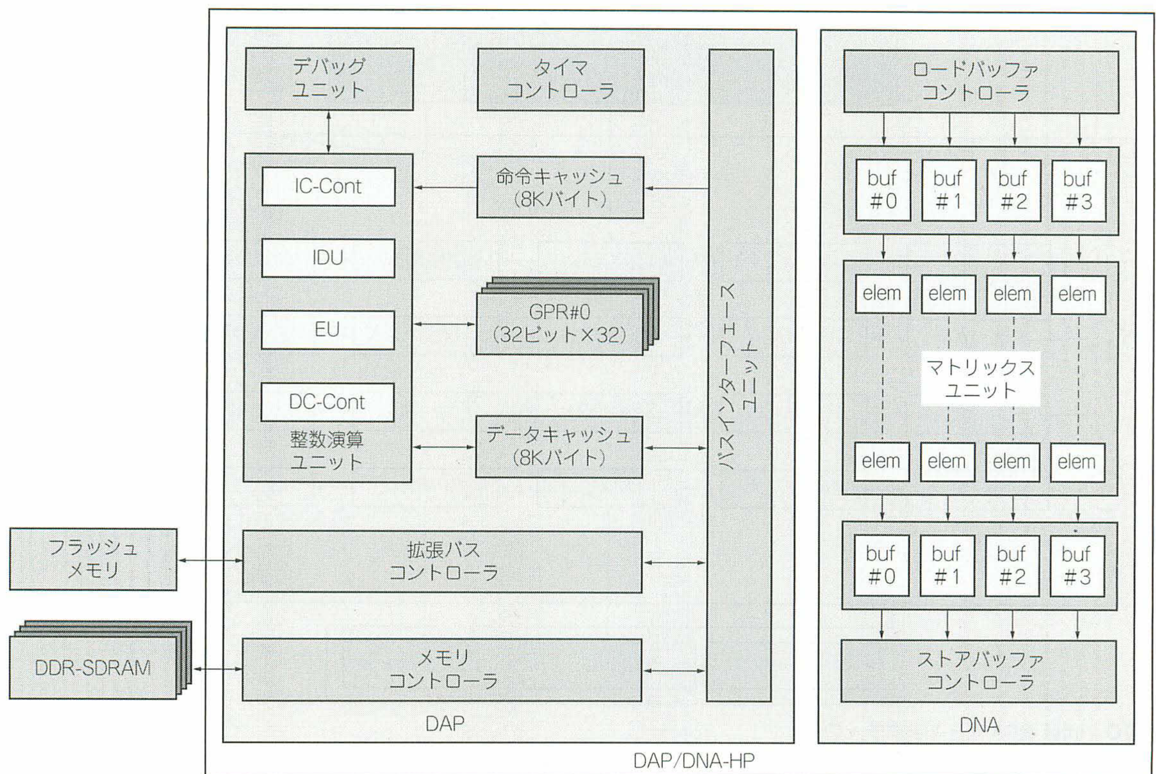
DAP/DNAのねらいは二つある．第一は，ハードウェア構造をほとんど意識せずに，ソフトウェア技術者が専用論理回路並みの性能を入手できることである．具体的にはC言語のソースコードから回路を生成することで，ソフトウェア技術者でも回路設計が可能になる．また，動作検証をシミュレーションで行っておけば，その動作が実チップでも保証される．第二は，一つのチップを種々の専用LSIに瞬時に置き換えられることである．たとえば，5mm角のチップが，従来の20mm角チップと同等の機能をもつという．

DAP/DNAが特徴的なのは，DNA内のマトリックスを制御する専用プロセッサであるDAPと呼ばれる専用CPUが存在する点である．このCPUにより，DNAマトリックスに対し，命令レベルによる切り替えと割り込み方式による切り替えの2種類が選択できる．また，DAPは16/32/48ビットの可変長命令のアーキテクチャを採用している．Xtensa内のCPUも可変長命令を使用しており，RISCでありながら命令コードの効率を重視する傾向がみてとれる．

DAP/DNAの第一弾は富士通が製造し，DAP/DNA-HPとしてサンプル出荷されている．DAP/DNA-HPは，高速ルータなどの主にネットワーク機器向けである．動作周波数は120MHzで消費電力は約5Wである．今回は，富士通のプロセス技術のライブラリ中で最高速なものを採用したが，低消費電力のライブラリを使えば，消費電力はさらに下げられる可能性がある．製造技術は0.18 μmルールのCMOSプロセスで，チップ面積は約15mm角である．

他プロセッサとDAP/DNAの処理性能の比較も発表されている．100MHz動作のDAP/DNAと700MHz動作のPentium Ⅲを比較した場合，AESの暗号化に要する処理時間はPentium Ⅲの1/13，FFT処理なら1/8，DCT処理は1/32という．最新のPentium4でも動作周波数は3GHzであるから，汎用チップでもさらに4倍程度の性能向上が見込まれるが，それでも，このような特定分野では，DAP/DNAのほうが高性能ということになる．

アイピーフレックスでは，要求される回路規模の巨大化に対応するため，DAP/DNA-2(2003年11月に発表予定)ではエレメント数を従来の148個から368個に増加させる．また，それでも回路に入りきらない場合



図M　DAP/DNAのブロック図

を想定して，別チップのDNAマトリックスと接続可能な専用インターフェースを内蔵するという(図N)．

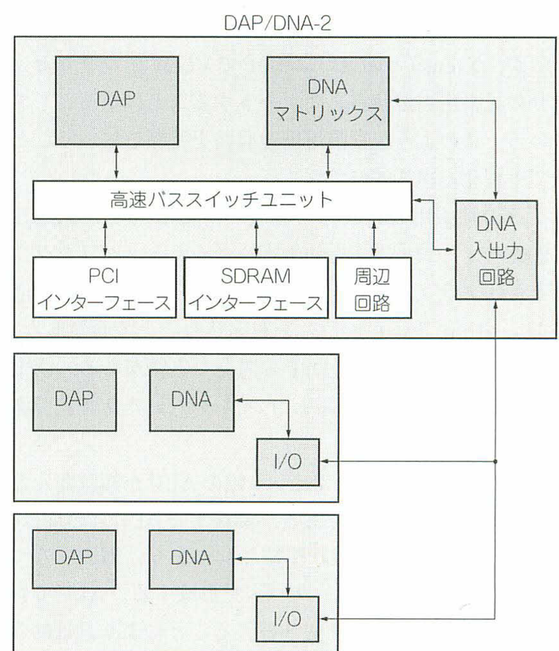
● 東芝のMePとElixent社のD-Fabrix技術

東芝は，アプリケーションに応じて機能や回路構成を柔軟に変更できるマイクロコントローラ「MeP (Media embedded Processor)」を開発している．簡単にいうと，MePは専用機能を有するCPUコアである．このCPUコアを複数個つなげて並列処理を行うことで一つの機能を実現する．

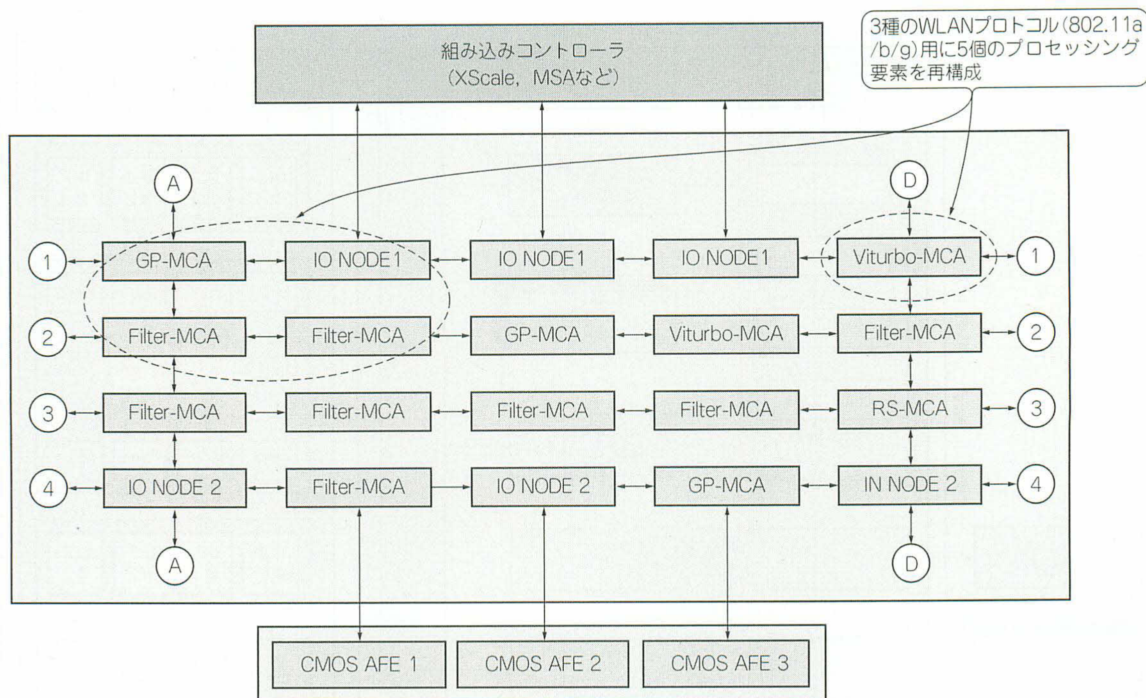
MePコアは，顧客のニーズに応じて命令セットや混載メモリ容量などをカスタマイズできる．そのアーキテクチャは東芝独自で，データ長は32ビット，命令長は16/32ビット可変長，汎用レジスタの数は16本である．

MePコアはそれ単体で使用されるのではなく，大別して4種類の拡張用周辺回路と組み合わせて専用LSIを形成する．拡張用周辺回路には，ユーザーが独自に定義できるユーザーカスタム命令(UCI)の実行回路，各種ハードウェアアクセラレータ回路，DSP，VLIW型コプロセッサがある．

MePの実装例として，東芝はMIPSアーキテクチャ



図N　DAP/DNA-2(他チップとの結合)



図O　Intel RCAアーキテクチャの例

TX49コアとMePを1チップに集積したMPEG-2デコーダチップ(TC81240TBG)を開発している．

基本的にはTensilica社のXtensaと同様な発想であるが，Xtensaとの違いはMePでVLIW型コプロセッサの拡張用周辺回路をサポートすることにあるとしている．このような拡張用周辺回路を蓄積していくことで差別化を図る戦略であろう．

その拡張用周辺回路の充実の一環として，東芝は2003年1月27日にElixent社のダイナミックリコンフィギュアラブル技術「D-Fabrix RAP(Reconfigurable Algorithm Processor)」を組み込むことで合意した．D-Fabrix RAPは専用DSPの開発で実績がある．CPUコア(MeP)も周辺回路もコンフィギュアラブルである点が興味深い．

D-Fabrix RAPは，4ビット幅のALUが複数並んだ並列型のプロセッサである．隣接するALUとの間は，4ビット幅のバスで相互接続されており，周囲の八つのALUと接続してデータパスを形成する．ALUで行う演算とバスの接続を切り替えることにより，目的の回路を生成する．なお，NECのDRPやアイピーフレックスのDAP/DNAとは異なり，1クロックでのリコンフィギュレーションには対応していない．リコンフィギュレーションの制御はD-Fabrix外部のCPUコアで行う．

● SONYのVME

SONYは，ダイナミックリコンフィギュアラブル技術を携帯型音楽プレーヤである「ネットワークウォークマンNW-MS70D」でいち早く実現した．UCバークレー校のJan Rabaey教授が研究していたPleiadesという技術の流れを汲むVME(Virtual Mobile Engine)と呼ばれるリコンフィギュアラブル技術を採用している．

音楽ファイルの復号化に従来はDSPを使用していたが，これに専用論理回路を用いれば消費電力を低減できることはわかっていた．しかし，音楽ファイルの形式にはいろいろな種類があり，その復号化処理を専用回路にするためには，その種類と同じ数だけの専用回路が必要になる．これを解決するための手法がVMEである．

音楽再生では1度に1種類の形式のファイルしか扱わない．再生する音楽ファイルの形式に応じて専用回路を再構成すれば複数の専用回路を装備する必要はない．

VMEと他のリコンフィギュアラブル技術との相違は，演算回路をオーディオ処理に特化することで低消費電力を図っている点である．他社がダイナミックリ

コンフィギュアラブル技術の汎用性を訴えるのとは対照的に，あえて応用分野を絞り込むことで回路の最適化を目指している．

VME技術を採用するオーディオLSIがCXR704060である．ATRAC3という形式の復号化時の消費電力は4mWであり，これはDSPを使用する場合の1/5であるという．

2003年4月には，VME技術のさらなる展開が始まった．ネットワークウォークマンに続き，CDウォークマン「D-NE1」と「D-NE9」にも採用された．使用している回路はNW-MS70Dと同一であるが，VMEを制御するソフトウェアが異なるという．また，従来のATRAC3やATRAC3plusのファイル形式のほかにMP3の再生機能も追加された．D-NE1では，低消費電力を特徴とするVME技術により世界最長の150時間の再生を実現した．

さらにSONYは，2004年に発表する携帯型ゲーム機PSP(PlayStation Portable)にVME技術を搭載することを発表している．動作周波数を166MHzに向上させ(NW-MS70Dでは22.58MHz)，映像処理にも対応させるという．

● IntelのRCA

2003年6月9日，Intelは東京都内のホテルにおいて記者説明会を開催し，同社が取り組んでいる無線技術について発表した．その中の一つにRCA(Reconfigurable Communication Architecture)と呼ばれる再構成可能な通信技術の発表があった．

現在の無線データ通信はいくつかの問題点を抱えている．そのため，各国の法令や複数の無線の規格に対応するデバイスの供給が必要となる．それを解決する技術の一つがRCAである．つまり，XScaleやDSPのような汎用プロセッサと専用ソフトウェアとRCAチップという単一の構成だけで，複数の無線技術に対応するという．

この説明会では，RCAの一例として，IEEE 802.11a/b/gといった三つの規格に対応するため，五つの要素を使って回路を再構成するメッシュ構成回路(**図O**)が紹介された．実際の再構成では，ファームウェアを変更して使用する要素の接続を切り替える．

なお，RCA技術はベンダー向けのものであり，ユーザーに公開する予定はないという．モジュールを別の規格に切り替える場合，認証作業が必要になるため，動的に再構成を行うことは事実上不可能であるからだ．

おわりに

コンフィギュアラブルプロセッサの概略について説明してきた．これらのプロセッサは技術的には興味深い．だが，コスト面から見ると，実際に組み込み制御に応用されるかというと，かなり微妙なものがある．しかし，ソフトウェア技術者がハードウェアを設計できるようになるという点では，将来的には有望かもしれない．システムコストよりも設計の短期間化が優先される場合は，さらに期待がもてるだろう．

参考文献

1) Tensilica，“Xtensa Product Brief”．
2) 佐藤友美，『10nsで演算器間の構成を書き換えるダイナミック・リコンフィギュアラブル技術を開発　実チップ寸法の4倍相当の機能を実現する』，日経エレクトロニクス，2003 1-6，pp.111-122.
3)『ソフトの開発が仮想回路を生むハードが瞬時に七変化　ダイナミック・リコンフィギュアラブル技術が産声』，日経エレクトロニクス，2002 11-18，pp.59-66.
4)『ソニーが一番乗り　ウォークマンに動的再構成　AV機器全般に同技術をヨコ展開へ』，日経エレクトロニクス，2003 1-20，pp.26-27.
5)『ソニーを道標に開発が加速　動的再構成はAV機器を越えてメモリ技術との融合で記憶と演算の一体化も視野』，日経エレクトロニクス，2003 9-15，pp.57-66.

第9章

浮動小数点演算を高速にこなすための FPUのしくみ

プロセッサが使う数値は整数ばかりではない．浮動小数点演算が必要になる場面も少なくない．浮動小数点演算には精度や丸め誤差の問題について注意を払う必要がある．ここでは，2進数で浮動小数点演算を行う場合の標準規格であるIEEE754について解説した後，実際の演算処理手順について具体例を示して解説する．

現在のコンピュータにおいて浮動小数点演算はほとんどの場合IEEE標準規格(IEEE Std 754-1985)に基づいて行われる．この規格が制定される前は，コンピューターメーカーが各社独自の浮動小数点形式を定義して浮動小数点演算ユニット(Floating Point Unit：FPU)を実装していたが，プログラムを他のコンピュータに移植する場合，データの互換性のなさが問題となっていた．IEEE Std 754-1985(通常IEEE754と呼ばれる)はIntelが提唱した浮動小数点の演算の仕様を叩き台として議論が重ねられ，1985年に制定された．そして現在，浮動小数点の標準規格といえばIEEE754のことを指すようになった．ほとんどすべてのMPUのFPUもこのIEEE754に準拠している．その意味でIEEE754は必須の教養であるということもできる．

本章ではIEEE754が定める浮動小数点のデータ型とその処理手順を解説する．

1 IEEE754とは

● 2進数で浮動小数点演算を行う場合の標準規格

IEEE754とは1985年3月21日にANSIで採択された『IEEE Standard for Binary Floating-Point Arithmetic』という2進数で浮動小数点演算を行う場合の標準規格である．規格書自体は20ページの薄い冊子で，具体的な浮動小数点演算の実装方式というよりも，実装方針を示している．この規格で規定されている項目は，

(1) 浮動小数点データの形式
(2) 加減乗除，平方根，剰余，比較演算
(3) 整数-浮動小数点データ間の形式変換
(4) 異なる浮動小数点形式間の変換
(5) 浮動小数点と10進ストリング間の形式変換
(6) 非数(NaN)を含む，浮動小数点例外と処理

である．しかし，一般に，浮動小数点演算の実装方式は確立されており，独自性を発揮する場は少ない．以降ではIEEE754の内容と一般的な演算の実装方式に関して説明していく．

● 浮動小数点データの形式

IEEE754標準は基本データ型として単精度(32ビット)と倍精度(64ビット)の浮動小数点データをサポートする．これ以外にもビット数の多い拡張精度(拡張単精度と拡張倍精度)が定義されているが実装依存なので割愛する．拡張精度の一つの実装である80ビット(あるいは96ビット)形式はなかばデファクトスタンダードだが，ここでは言及しない．単精度の浮動小数点形式は**図1(a)**に示すように，24ビットの符号付き仮数部(s＋f)と，8ビットの指数部(e)で構成される．倍精度の浮動小数点形式は**図1(b)**に示すように，53ビットの符号付き仮数部(s＋f)と，11ビットの指数部(e)で構成される．浮動小数点データは，次の三つの領域により成り立っている．

(1) 符号ビット：s
(2) 指数部　：e＝E＋バイアス値
　(単精度→0x7f；倍精度→0x3ff)
(3) 仮数部　：$f = .b_1b_2b_3b_4$……
　(小数点以下第1位以下の値)

これらの情報で，

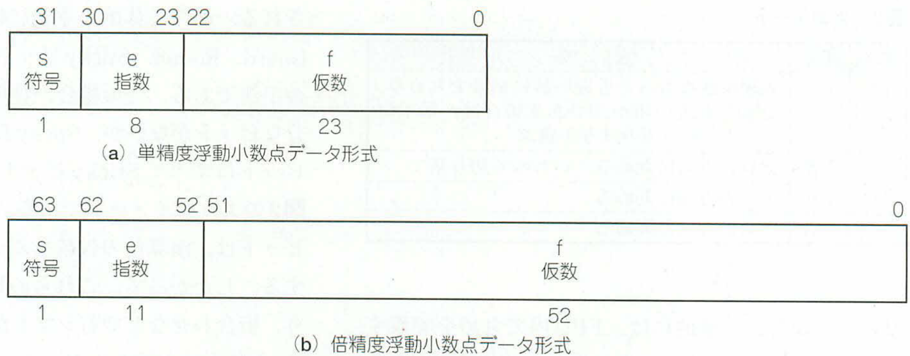


図1 浮動小数点のデータ形式

$\pm 1.b_1b_2b_3b_4\cdots\cdots \times 2^E$(2のE乗)

のデータを表す．浮動小数点のデータ形式では整数部の1を省いて表現している．このような形式で表現できる数を正規化された数(ノーマル数)と呼ぶ．この1を省く表現をケチ表現ということもある．80ビット以上の拡張精度はケチ表現ではないが，ここでは説明しない．

一方，整数部が0である，

$\pm 0.b_1b_2b_3b_4\cdots\cdots \times 2^E$(2のE乗)

のようなデータ形式を非正規化数(デノーマル数)と呼ぶ．また，この他に，IEEE754では非数(NaN)，無限大，ゼロという特殊なデータ型を定義している．

NaN(Not a Number)とは，指数部(e)が最大値で，仮数部(f)が0以外のデータである．さらに，fの最上位ビット(小数点以下第1位)が1のときをSignaling-NaN(SNaN)，0のときをQuiet-NaN(QNaN)と呼ぶ．NaNは浮動小数点演算において無効な演算を行った場合の結果として用意された記号で数ではない．QNaNを用いた演算の結果はQNaNとなり，一度QNaNが発生するとのちのちの演算を通じて伝播していく．SNaNはユーザーが意図的に与えることのできるNaNであり，それを用いた演算は無効浮動小数点演算例外(後述)を発生する．もし無効浮動小数点演算例外の発生が禁止されている場合はQNaNを結果とする[注]．

無限大とは，指数部(e)が最大値で，仮数部(f)が0のデータである．数学でいうところの無限大と同様の意味をもつ．無限大には，符号(s)によって正の無限大と負の無限大がある．

ゼロとは，指数部(e)と仮数部(f)が0のデータである．数学でいうところの0と同様の意味をもつ．符号(s)によって正のゼロと負のゼロがある．比較演算においては，正のゼロと負のゼロは等しいものとされる．

表1 ±1.0(ノーマル数の代表)，QNaN，SNaN，±無限大，±ゼロの16進数表現

+1.0	0x3f800000 0x3ff0000000000000	(単精度) (倍精度)
−1.0	0xbf800000 0xbff0000000000000	(単精度) (倍精度)
QNaN	0x7fbfffff 0x7ff7ffffffffffff	(単精度，一例) (倍精度，一例)
SNaN	0x7fffffff 0x7fffffffffffffff	(単精度，一例) (倍精度，一例)
+∞	0x7f800000 0x7ff0000000000000	(単精度) (倍精度)
−∞	0xff800000 0xfff0000000000000	(単精度) (倍精度)
+0	0x00000000 0x0000000000000000	(単精度) (倍精度)
−0	0x80000000 0x8000000000000000	(単精度) (倍精度)

以上の，±1.0(ノーマル数の代表)，QNaN，SNaN，±無限大，±ゼロを16進数で表現すると，それぞれ**表1**のようになる．

● 丸めモード

丸めとは，表現する数値があるデータ範囲内に入るように仮数部のLSB(Least Significant Bit)に対して切り上げまたは切り捨てを行い，近似値を求めることである．IEEE754はRN(Toward Nearest)，RZ(Toward Zero)，RP(Toward Plus Infinity)，RM(Toward Minus Infinity)の4種の丸めモードを定義している．丸めモードの意味を**表2**に示す．

ただし，IEEE754では丸めの実装方式は規定してい

注：SNaNとQNaNを区別する形式はIEEE754では規定されていない．ここでは，一般的な形式にしたがった．

表2　丸めモード

RM	記号	意　味
00	RN	表現可能なもっとも近い値に結果を丸める．もっとも近い値が二つある場合は，最下位ビットが0の値のほうを選ぶ
01	RZ	ゼロの方向に丸める．いわゆる切り捨て
10	RP	＋∞の方向に丸める
11	RM	－∞の方向に丸める

ない．しかし，一般的には，FPU内で丸めを実現するために，浮動小数点データに丸め用の補助ビットを3ビット追加して演算を行う．それがGuard，Round，Stickyビットである．GuardとRoundについては単に演算精度を増やすだけであるが，Stickyには特別な意味がある．演算の途中結果で右シフト(桁合わせのため)を行った場合，シフトアウトされるあふれビットのすべての論理和(つまり蓄積した値)がStickyになる．

IEEE754において浮動小数点の仮数部を扱う時，あたかも無限の精度があるかのように扱うことが要請される．その具体的な手法には言及されていないが，Guard，Round，Stickyビットがそれを実現する一つの手法である．この場合，物理的には3ビットしか余分なビットがないが，Sticky以下の重みの(右にある)ビットはすべてStickyビットに代表される．それは**図2**のようなイメージである．Guard，Round，Stickyビットは，演算前の仮数を3ビット左シフトして生成する．したがって，これらのビットの初期値は0であり，桁合わせなどで右シフトが発生するときに変化する(上位の値が繰り下がってくる，ただしStickyビットはシフトアウトされる値との論理和)．

丸めは，結果の符号，仮数の最下位ビット，Guard，Round，Stickyビットの値と丸めモードとの関係で行われる．具体的には**図3**に示すような関係を見て，仮数の切り上げ，切り捨てを決定する．

なお，実装によっては，GuardビットとStickyビットのみで精度を維持する場合もある．

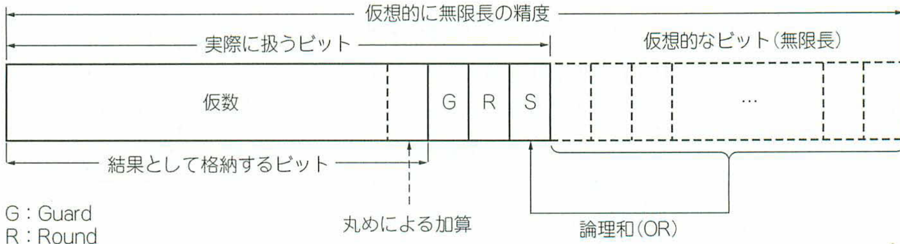


図2　Guard，Round，Stickyビットの意味

丸めモード	符号	仮数部LSB	Guard	Round	Sticky	切り上げ切り捨て	
RN	×	×	0	×	×	↓	R：LSBに加算する値
	×	×	1	1	×	↑	
	×	×	1	0	1	↑	R=Guard ＆ (LSB ｜ Round ｜ Sticky)
	×	1	1	0	0	↑	
	×	0	1	0	0	↓	
RZ	×	×	×	×	×	↓	R=0
RP	1	×	×	×	×	↓	
	0	×	1	×	×	↑	
	0	×	0	1	×	↑	R=(˜Sign) ＆ (Guard ｜ Round ｜ Sticky)
	0	×	0	0	1	↑	
	0	×	0	0	0	↓	
RM	0	×	×	×	×	↓	
	1	×	1	×	×	↑	
	1	×	0	1	×	↑	R=Sign ＆ (Guard ｜ Round ｜ Sticky)
	1	×	0	0	1	↑	
	1	×	0	0	0	↓	

×：don't care

図3　丸めの実現

● 符号ビットに関する注意

IEEE754では結果がゼロになる場合の符号について厳密な規定がある．比較演算において，－0と＋0は同じものとみなされるので，ここまで神経質になる理由はよくわからない．ともかく，IEEE754では次のように記述されている．

『異符号である二つのオペランドの和(あるいは同符号のオペランドの差)は正確にゼロである．和(あるいは差)の符号は，ほとんどの丸めモードでプラス(＋)となる．ただし，－∞方向の丸めモードにおいてはマイナス(－)となる．しかし，$x+x=x-(-x)$の場合は，xがゼロであっても，結果はxの符号と同じになる．また，$\sqrt{-0}$の結果は－0である．』

これは，オペランドがゼロでない場合，丸めモードが－∞方向以外の結果は＋0，－∞方向では－0となることを意味する．ただし，加減算の両方のオペランドがゼロの場合は，丸めモードにかかわらず，第1オペランドの符号が結果の符号になる．つまり，$(+0)+(+0)=+0$，$(+0)-(-0)=+0$，$(-0)+(-0)=-0$，$(-0)-(+0)=-0$である．

2 浮動小数点演算命令の処理手順

● 浮動小数点演算に共通する処理手順の概略

ここでは，浮動小数点演算全体に共通する処理手順の概略を示す．FPUは次に示す手順をハードウェアで実現する．ハードウェアの概略を図4に示す．

(1) オペランド処理

命令によって指定された，一つまたは二つのソースオペランドを，符号，指数，仮数の部分に分解するとともに，特殊データ型のいずれに属するかを判定する．もし，無限大またはNaNと判定された場合は，そのための処理(後述)を行う．もし，デノーマル数と判断された場合は，指数の精度を無限大と考えて，演算前に正規化が実行される．

(2) 演算処理

(1)で得られたオペランドを基に演算が実行される．演算はGuard-Round-Stickyと呼ばれる表現形式上での最下位ビットよりも小さな位取りのビット群を含めて実行される．無効浮動小数点演算例外(Invalid Floating Operation Exception)，浮動小数点ゼロ除算

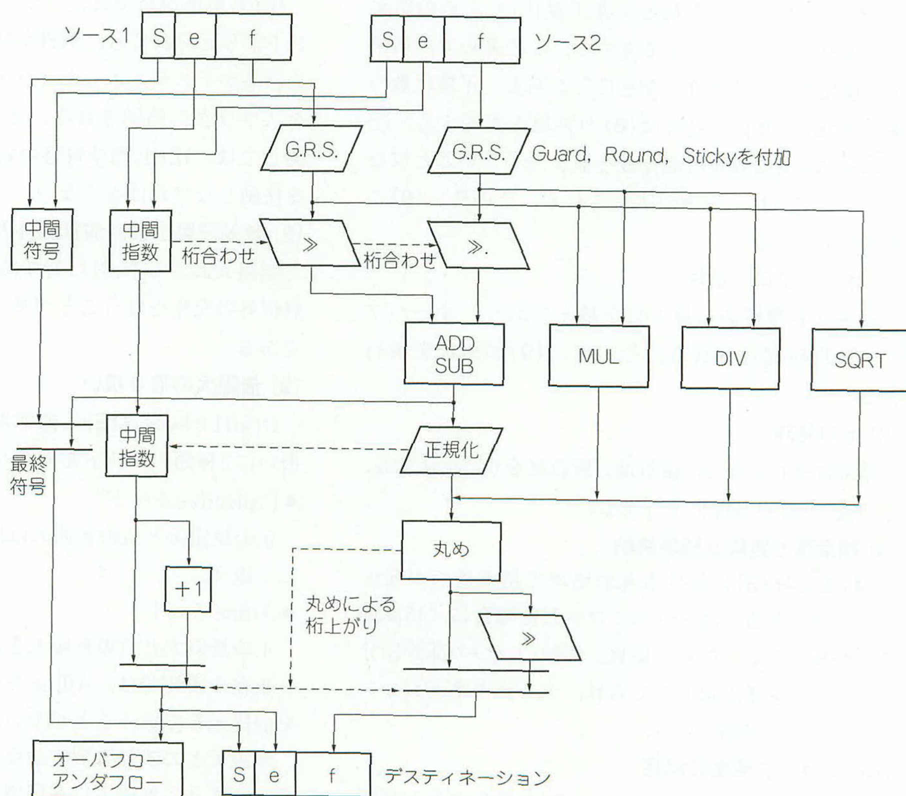


図4 FPUの構成例

例外(Floating Point Zero Divide Exception)は，例外が検出された時点で例外処理が実行される．

比較演算は，「<」，「=」，「>」，「比較不能」というフラグを生成する．比較不能とは片方，または両方のソースオペランドがNaN(SNaNまたはQNaN)である場合である．

(3) 結果の正規化

(2)で得られた結果を正規化する．正規化の結果，仮数部(Guard-Round-Stickyを含めて)が0となった場合は，(8)の処理を実行する．

(4) 結果の丸め

(3)で得られた結果を，丸めモードにしたがって丸める．(3)で得られた結果も(6)の処理で使用する可能性があるので保存しておく．

(5) 結果の分類

(4)の結果，得られた指数部の値が0以下の場合は(6)，最大値より大きい場合は(7)の処理を実行する．このいずれでもない場合,結果は正規化数とみなされ，(9)の処理を実行する．

(6) アンダフロー処理

(3)で得られた結果を，指数部の値が1となるようにスケーリング(指数部の値を1増加すると，仮数部は1ビット右シフトされる：非正規化)し，その結果を丸めモードにしたがって丸める．この丸めで，結果が正規化数またはゼロになることがある．正規化数の場合は(9)，ゼロの場合は(8)の処理を実行する．それ以外の場合は表現可能な最小値より小さいことになり，アンダフロー処理が実行される．その後，(9)の処理を実行する．

(7) オーバフロー処理

結果は表現可能な最大数を越えており，オーバフロー処理が実行される．その後，(9)の処理を実行する．

(8) ゼロ処理

結果はゼロになる．指数部，仮数部を0に設定する．その後，(9)の処理を実行する．

(9) 精度落ち処理と結果格納

(4)または(6)における丸め処理で精度落ちが発生した場合，あるいはオーバフローした場合に「精度落ち」が発生する．符号，指数，仮数の三つの部分に分かれている結果が組み立てられ，デスティネーションオペランドに格納される．

(10) トラップ発生の判定

(6)，(7)，(9)においてトラップが発生すると判断された場合(トラップ条件を満たし，かつトラップが許可されている)は，所定の例外ハンドラに制御を移す．

以上が演算の処理手順であるが，現実の実装においてはハードウェアの簡略化のために詳細な処理を省略することもある．たとえば，

- ソースオペランドがデノーマル数，無限大，NaNの場合は例外を発生させて例外ハンドラでのソフトウェアによる処理に任せる．
- 演算結果がデノーマル数になる場合は無条件にゼロに丸める．
- 演算の途中の指数の値でオーバフローやアンダフローを検出する．たとえば，丸めの直前や直後の指数の値でオーバフローやアンダフローを決定してしまう．

などが普通に行われる，具体的にどうなっているかを知るには，各MPUのマニュアルを参照する必要がある．

● オペランドが無限大またはNaNの場合の処理

IEEE754は無限大やNaNに対する演算も規定している．これは次のような点を考慮して行われる．

(1) 演算結果の格納

IEEE標準規格では，いかなる種類のソースオペランドが与えられても，例外が発生しなければ，何らかの結果がデスティネーションオペランドまたはステータスフラグに格納される．とくに，無限大とNaNに関しては，IEEE標準規格の定める演算結果(定数値)を格納しなければならない．

(2) 無効浮動小数点演算例外の発生

無限大およびNaNに対する演算は，無効浮動小数点例外の発生を伴うことが多い．この点の考慮も必要である．

(3) 無限大の取り扱い

Draft1.0以前のIEEE標準規格では，無限大の取り扱いに2種類のモードをもたせている．

- Projectiveモード

正の無限大と負の無限大は区別されず等価なものとして扱う．

- Affineモード

正の無限大と負の無限大を区別する．

現在の実装では，Affineモードをサポートすることを前提としていることが多い．

無限大との演算は数学的なイメージと同じ結果になる．つまり，無限大に有限値を加減算しても無限大で

あるし，正の無限大に正の無限大を加算しても正の無限大である．ただし，正の無限大から正の無限大を減算するような，数学的にも意味のない演算は，無効浮動小数点演算となる．

3 浮動小数点演算で発生する例外

● IEEE754での例外の定義

IEEE754は次の五つの例外が「検出されるべき」として定義されている．

- 無効浮動小数点演算例外
  (Invalid Floating Operation Exception)
- 浮動小数点ゼロ除算例外
  (Floating Zero Divide Exception)
- 浮動小数点オーバフロー例外
  (Floating Overflow Exception)
- 浮動小数点アンダフロー例外
  (Floating Underflow Exception)
- 浮動小数点精度落ち例外
  (Floating Precision Exception)

これらの例外について，次の点を考慮しなければならない．

(1) 例外発生条件

五つの例外事象の発生条件が定められている．

(2) 例外事象通知

五つの例外事象の各々に対し，それらを独立に通知することのできる五つの条件フラグが必要である．

(3) トラップ発生指定

五つの例外事象の各々に対し，それらの発生に伴いトラップを発生させるか否かを独立に指定することのできる五つのマスクフラグが必要である．例外事象が発生しても，マスクフラグによってトラップの発生が禁止されている場合は例外にならず，所定の値がデスィネーションオペランドに格納されて，演算は続行される．

(4) 結果格納

五つの例外事象の各々に対し，トラップの有無に対応した値をデスティネーションに格納する．

(5) トラップ命令

五つの例外事象の発生に伴うトラップを，保存された例外事象の履歴とトラップ発生の指定に基づいて発生させる専用命令をもつ．

● 例外の内容

次に五つの例外について説明する．

(1) 無効浮動小数点演算例外（Invalid Floating Operation Exception）

この例外は無限大やNaNとの演算，数学的に意味のない演算（0/0など）を行った場合や，整数への形式変換時の変換不正の場合に発生する．具体的には以下のような場合がある．

- (+∞)+(−∞)，(−∞)−(−∞)
- ±0×±∞
- ±0÷±0，±∞÷±∞
- オペランドに順序付けがない場合での大小比較
- オペランドにSNaNを含む算術命令
- オペランドがSNaNの場合の比較と浮動小数点への変換（オペランドがQNaNの場合はその値を伝播するだけで例外にはならない）
- オペランドが負の平方根

この例外において，トラップが発生する場合はデスティネーションは変化しないが，トラップが発生しない場合はデスティネーションにはNaN（QNaN）が格納される．

(2) 浮動小数点ゼロ除算例外（Floating Zero Divide Exception）

ゼロ除算例外は，正規化数あるいはデノーマル数（ゼロでない有限値）を0で除算する場合に発生する．

トラップが発生する場合，デスティネーションは変化しないが，トラップが発生しない場合，デスティネーションには無限大が格納される．

(3) 浮動小数点オーバフロー例外（Floating Overflow Exception）

オーバフロー例外は，演算命令や変換命令において，結果の丸め後の値が有限であり，デスティネーションで表現可能な最大数よりも大きくなった場合に発生する．

トラップが発生する場合はデスティネーションは次のようになる．

- 変換命令の場合は変化しない
- 演算命令の場合は補正された指数部（後述）と仮数部が格納される

トラップが発生しない場合にはデスティネーションには次の値が格納される．

- 丸めモードがToward Zeroのとき，表現可能な絶対値が最大の正規化数
- 丸めモードがToward −∞で，かつ結果の符号が+のとき，正の表現可能な最大数
- 丸めモードがToward +∞で，かつ結果の符号が

表3　オーバフロー/アンダフロー例外発生時の補正値

	単精度	倍精度
オーバフロー	−192(−0xC0)	−1536(−0x600)
アンダフロー	+192(+0xC0)	+1536(+0x600)

−のとき，負の表現可能な最大数(絶対値が)

- その他の場合は無限大

(4) 浮動小数点アンダフロー例外(Floating Underflow Exception)

アンダフロー例外は，転送命令，演算命令，変換命令において，結果を丸めた値が，デスティネーションで正規化数として表現可能な最小数より小さくなった場合に発生する．

トラップが発生する場合，デスティネーションは次のようになる．

- 変換命令の場合は変化しない
- 演算命令の場合は補正された指数部(後述)と仮数部が格納される

トラップが発生しない場合，デスティネーションは丸めモードによってデノーマル数またはゼロが格納され，その丸めによって精度落ちがあった場合のみアンダフローフラグがセットされる．

- 丸めモードがToward Zeroのときゼロ
- 丸めモードがToward −∞で，かつ結果の符号が−のとき，正の表現可能な最小数
- 丸めモードがToward +∞で，かつ結果の符号が+のとき，負の表現可能な最小数(絶対値が)
- その他の場合はゼロ

(5) 浮動小数点精度落ち例外(Floating Precision Exception)

精度落ち例外は，演算命令や変換命令における結果の丸めにおいて，その値が不正確(Inexact)になった場合に発生する．精度落ち例外は，ほかのすべてのトラップが発生しなかった場合に発生する．

トラップの発生の有無にかかわらず，精度落ちフラグがセットされ，デスティネーションには丸めた結果が格納される．

精度落ち例外は，丸め前または丸め後のGuard，Round，Stickyビットのいずれかが1であった場合に発生する．ということは，ほとんどの演算(仮数部の右シフトがある場合)において精度落ち例外となる．あるいは，演算においてオーバフローやアンダフローが発生する場合や，型変換で結果が目的のビット数に収まらない場合に精度落ち例外となる．

● オーバフロー例外，アンダフロー例外発生時の結果

IEEE754では，オーバフロー例外，アンダフロー例外，精度落ち例外発生時には，結果を正しく丸めて返すことが要請される．これは，結果がデスティネーション形式に合致するか否かを示す情報が付加される．精度落ち例外に関しては，結果は自明である(得られた結果が精度落ちしていると表明しているだけ)．しかし，オーバフロー例外，アンダフロー例外に関しては，なるべく精度を保持するためにいろいろな手法が考案されている．

たとえば，初期の実装には**表3**に示す補正バイアス値を加算して指数部を補正した値を返すものもあった．ここで示された値によって指数部を補正すると，結果は必ず正規化数として表せる．

しかし，実際のFPUでは補正した値ではなく近似値がほしい場合が多いので，オーバフロー時は無限大または最大ノーマル数，アンダフロー時は負の無限大またはゼロを結果とすることが最近の流行である．これは，いわゆる飽和処理である．

● 遅延トラップ発生

上述の五つの例外事象の発生にともなうトラップは，例外の発生した命令の直後に発生する．MPUによっては，保存された例外事象の履歴とトラップ発生の指定に基づいて例外を発生させる命令がある．つまり，その命令を実行した時点で例外が発生していればトラップする．これによりトラップを命令単位より大きな単位(たとえば手続き単位)で発生するように制御できる．

● 例外の発生

通常のプロセッサでは，上述の五つの例外の発生を示すための5種のフラグと，それに対応する5種のマスクがある．たとえば，

- 無効浮動小数点演算例外　…　V
- 浮動小数点ゼロ除算例外　…　Z
- 浮動小数点オーバフロー例外　…　O
- 浮動小数点アンダフロー例外　…　U
- 浮動小数点精度落ち例外　…　I

というフラグが定義されているとし，対応するマスクを，それぞれMV，MZ，MO，MU，MIとする．この場合，実際に例外によるトラップが発生するのは

(V and MV) or (Z and MZ) or (O and MO) and (U and MU) and (I and MI)

の値が1になる場合である．なお，各フラグとマスク

はソフトウェアで値を設定できる．また，V，Z，O，U，Iの各フラグは演算結果によって設定し直されるのではなく，それまでの値を蓄積していく．つまり，

新フラグ ← (旧フラグ) OR (演算結果のフラグ)

というイメージである．したがって，フラグの値がいったん1になると，ソフトウェアで0にクリアするまで1のままである．

4 演算精度について

● 演算精度

IEEE754における浮動小数点演算は，浮動小数点データが無限の精度をもっているように行われる．このため中間の演算の精度をどのような精度で行おうとも結果は同じにならなければならない．

たとえば，単精度の浮動小数点演算をそのままの精度で実行しても，倍精度で実行しても，あるいは内部的に80ビット(拡張精度)で実行しても，結果を丸めて単精度にした場合はすべて同じ結果にならなければならない．これは，Guard，Round，Stickyビット(特にStickyビット)を付加することで実現できるとされている．ただし，積和演算のように，乗算の後に加算という2回以上の演算を実行するときは，実行速度を上げるために，1回目の演算結果を丸めずにそのまま2回目の演算を行う場合がある．このような場合は，内部の演算精度によって結果が異なることがある．

また，逆数を計算する場合も，データを1.0から除算するだけなので，どのように計算しても同じ結果になるはずであるが，実装では結果の一致性を保証しないことが多い．これは逆数の高速近似アルゴリズムで演算を実行する場合を考慮してのものである．

● (a＋b)＋c≠a＋(b＋c)の話

浮動小数点演算には丸め誤差が付き物である．このため，演算順序を変更すると結果が異なることがある．数値演算の理論によれば，丸め誤差を最小にするために演算は加算をまとめ，位の大きさの似た数値から(それも小さいものから)順次加えていくこと推奨される．位の大きさの似た数値どうしの減算は，有効数字が大幅に減少するので，できるだけ避けなければならない．あるいは，丸め誤差を最小にするために，ピボットと呼ばれるオフセットをわざわざ減じて位の大きさを揃えることも常識である．このような意図をもって書かれたプログラムの順序を変更することは許されない．

しかし，これは浮動小数点演算をコンパイラが最適化するときに問題になってくる．整数には丸め誤差がないので，自由自在に順序の入れ替えが可能だったが，浮動小数点演算の順序を入れ替えるとプログラマの意図しない結果を生じることがある．このへんは最適化を行うに際してのトレードオフとなっている．

5 浮動小数点演算処理の実際

● 実際に計算方法

以降では，加算，減算，乗算，除算，平方根，32ビット整数から単精度浮動小数点への変換，32ビット整数から倍精度浮動小数点への変換，単精度浮動小数点から32ビット整数への変換，倍精度浮動小数点から32ビット整数への変換，比較に関して処理を実現する手順を示す．簡単のため，多くの場合は入力はノーマル数であることを仮定している．特殊データ型の処理はここでは言及しない．

浮動小数点演算の手順を大雑把に示すと，

- アンパック(符号，指数部，仮数部に分解)
- 演算
- 丸め
- リパック(符号，指数部，仮数部を結合)

という流れになる．アンパック，丸め，リパックについてはほとんどの演算について共通である．アンパックとリパックの処理を**図5**に示す．

以降の処理では，アンパック後のソース1，ソース2の符号，指数，仮数(1を付加したもの)を，それぞれ，

fs_sgn，fs_exp，fs_man，

ft_sgn，ft_exp，ft_man，

と仮定する．また，リパック直前の符号，指数，仮数を，それぞれ，

fd_sgn，fd_exp，fd_man，

と仮定する．また，仮数(fs_man，ft_man，fd_man)にはGuard，Round，Stickyの3ビットが下位に付加されているものとして読んでほしい．指数はバイアス(倍精度：0x3ff，単精度：0x7f)付きである．

● 加算

まず，10進数で加算のアルゴリズムを検討しよう．二つの浮動小数点を，

$a.bbbb \times 10^c \quad (a \neq 0)$

$d.eeee \times 10^f \quad (d \neq 0)$

とする(小数点以下は4桁と仮定)．数学で習ったよう

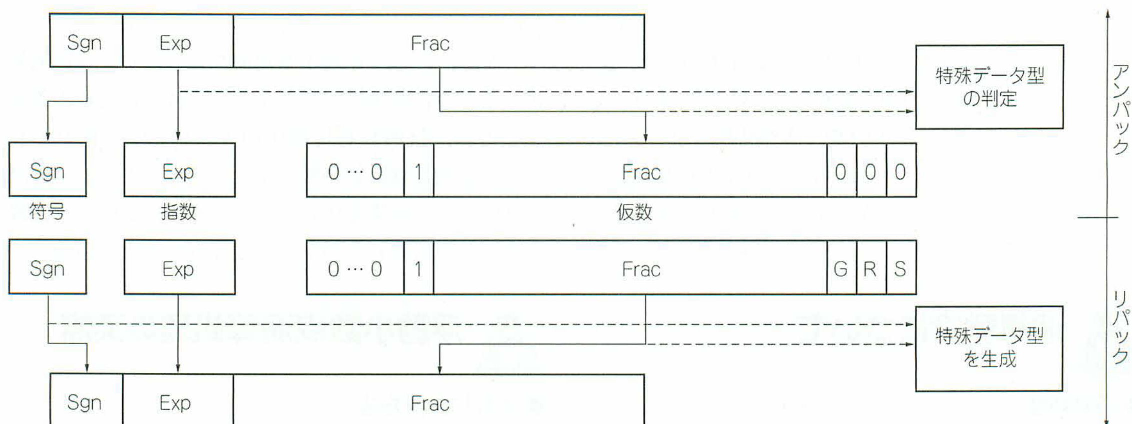


図5　アンパック/リパック処理

に，指数形式で表現された数の加算は指数部を等しくして行う．指数の大きいほう，小さいほうのどちらに合わせてもいいのだが，結果は指数の大きいほうと，オーダー的に，ほぼ等しくなるので，大きいほうに合わせる．指数の小さいほうを大きいほうに合わせるためには指数の差だけ仮数部を右シフトする．いまの場合，10進数なので，

$10^{-(c-f)}$　($c > f$と仮定)

を仮数部に乗算することに等しい．その結果，二つの数は，

$a.bbbb \times 10^{c}$

$0.eeee \times 10^{c}$

となる．これにより加算結果は，

$g.gggg \times 10^{c}$　($g.gggg = a.bbbb + 0.eeee$)

となる．また，仮数が異符号(今は両方とも正としているが)の場合は，

$0.0hhh \times 10^{c}$　($0.0hhh = a.bbbb - 0.eeee$)

という結果になる．この場合は正規化するために，仮数部を10^{2}倍(これは左シフトに相当)して，

$h.hh00 \times 10^{(c-2)}$

という結果になる．10進数の場合はこれで終わりである．

しかし，2進数の場合は，加算後の仮数の整数部(小数点の左)が桁上がりする可能性がある．つまり，無理やり10進数でいえば，

$ii.jjjj \times 10^{c}$

という形式になることがある(10進数で計算する場合はあり得ない)．この場合，正規化するためには仮数部を10^{-1}倍することになり，結果は，

$i.ijjj \times 10^{(c+1)}$

となる．以上の計算を2進数で行う(**リスト1**)．

加算のアルゴリズムは，指数部の値が大きいほうに，小さいほうを桁合わせして加算するものである．アンパックとリパックを除く手順を次に示す．特殊データ型の処理や，オーバフロー，アンダフローの検出処理は省略してある．

なお，加算においてはソースオペランドの順序を入れ替えても結果は変わらない．このため，たとえば，第2ソースオペランドの仮数部のほうが小さくなるように，あらかじめソースオペランドを入れ替えておけば，桁合わせ用の右シフタは，第2ソースオペランド用に一つあればよい．**リスト1**の説明では，簡単のために，それぞれのソースオペランドごとに二つの右シフタがあるものとしている．

また，仮数部の加算処理をまともに実装すると，ソースオペランドの符号の組み合わせによっては，1クロックで，

符号反転 → 加算 → 符号反転

と，非常に重い処理をしなければならない．これは，二つのソースオペランドの符号と大小関係と結果の符号を考慮すると，1回の加算または減算で処理することが可能である．しかし，アルゴリズムをわかりにくくするので，今回はこの手法を説明していない．

ところで，結果の符号は，基本的には仮数部の加算結果と同じでよいが，結果がゼロになる場合は，上述したような規則がある．ここではこの処理は省略してある．

● **減算**

減算は，第2ソースオペランドの符号を反転して加算と同じ処理をするのみでよい．

リスト1 加算処理

```
if(fs_exp < ft_exp) then
    fd_exp = ft_exp;
    ts_man = Sticky付き右シフト fs_man を (ft_exp-fs_exp) ビット;
else
    fd_exp = fs_exp;
    ft_man = Sticky付き右シフト ft_man を (fs_exp-ft_exp) ビット;
endif
```

(a) 桁合わせ

```
if(fs_sgn == 1) then /* 負なら */
    fs_man = -fs_man;
endif
if(ft_sgn == 1) then /* 負なら */
    ft_man = -ft_man;
endif
fd_man = fs_man + ft_man;
if(fd_man < 0) then /* 絶対値 */
    fd_sgn = 1;
    fd_man = -fs_man;
else
    fd_sgn = 0;
endif
```

(b) 加算

```
fd_man = {0...0, fd_man} /* 64ビット長を仮定 */
clz    = fd_man をMSB(bit63)から下位に連続する0の数を計数;
         /* 1の位置を検出 */
if(倍精度) then
    if(clz == 7) then /* 桁上がりした */
        fd_exp = fd_exp + 1;
        td_man = Sticky付き右シフト fd_man を 1ビット;
    else
        fd_exp = fd_exp - (clz-8);
        fd_man = 左シフト fd_man を (clz-8)ビット;
    endif
else
    if(clz == 36) then /* 桁上がりした */
        fd_exp = fd_exp + 1;
        td_man = Sticky付き右シフト fd_man を 1ビット;
    else
        fd_exp = fd_exp - (clz-37);
        fd_man = 左シフト fd_man を (clz-37)ビット;
    endif
endif
```

(c) 正規化

```
fd_man = fd_man を丸めモードに従って丸める;
if(倍精度) then
    if(fd_man[56] == 1) then /* 桁上がりした */
        fd_exp = fd_exp + 1;
        td_man = 右シフト fd_man を 1ビット;
    endif
else
    if(fd_man[27] == 1) then /* 桁上がりした */
        fd_exp = fd_exp + 1;
        td_man = 右シフト fd_man を 1ビット;
    endif
endif
```

(d) 丸め

● 乗算

例によって，10進数で乗算のアルゴリズムを検討しよう．二つの浮動小数点を，

$a.bbbb \times 10^{c} \quad (a \neq 0)$

$d.eeee \times 10^{f} \quad (d \neq 0)$

とする(小数点以下は4桁と仮定)．乗算の場合の結果は簡単で，

$(a.bbbb \times d.eeee) \times 10^{(c+f)}$

となる．このときの仮数部は，

g.gggggggg

または，

hh.hhhhhhhh

となる．正規化をすると，それぞれ，

$g.gggg \times 10^{(c+f)}$

または，

$h.hhhh \times 10^{(c+f+1)}$

となる．つまり，結果の仮数部は仮数どうしの乗算結果の上位をそのまま取り出しただけである．仮数部を固定小数点として乗算すると考えると，5桁と5桁を乗算して9桁または10桁になる乗算結果を，4桁または5桁分右シフトすることに等しい．

乗算は，仮数部どうしを乗算して，結果を右シフト(Sticky付き)して正規化して丸めればよい．結果の符号は，ソースオペランドの符号の排他的論理和になる．結果の指数は，ソースオペランドの指数(バイアスを除く)を加算したものである．アンパックとリパックを除く手順を**リスト2**に示す．特殊データ型の処理や，オーバフロー，アンダフローの検出処理は省略してある．

なお，丸め処理は加算の丸め処理と同じである．

● 除算

まずは10進数で乗算のアルゴリズムを検討しよう．二つの浮動小数点を，

$a.bbbb \times 10^{c} \quad (a \neq 0)$

$d.eeee \times 10^{f} \quad (d \neq 0)$

とする(小数点以下は4桁と仮定)．除算の場合の結果は，

$(a.bbbb \div d.eeee) \times 10^{(c-f)}$

となる．このときの仮数部は，

g.gggggggg....

または，

0.hhhhhhhh....

となる．正規化をすると，それぞれ，

$g.gggg \times 10^{(c-f)}$

または，

$h.hhhh \times 10^{(c-f-1)}$

リスト2　乗算処理(乗算と正規化)

```
fd_sgn = fs_sgn xor ft_sgn;
fd_man = fs_man × ft_man;
if(倍精度) then
    fd_exp = fs_exp + ft_exp - 0x3ff;
    fd_man = Sticky付き右シフト fd_man を 55ビット;
    if(fd_man[56] == 1) then /* 桁上がりした */
        fd_exp = fd_exp + 1;
        td_man = 右シフト fd_man を 1ビット;
    endif
else
    fd_exp = fs_exp + ft_exp - 0x7f;
    fd_man = Sticky付き右シフト fd_man を 26ビット;
    if(fd_man[27] == 1) then /* 桁上がりした */
        fd_exp = fd_exp + 1;
        td_man = 右シフト fd_man を 1ビット;
    endif
endif
```

リスト3　除算処理

```
fd_sgn = fs_sgn xor ft_sgn;
if(倍精度) then
    fd_exp = fs_exp - ft_exp + 0x3ff;
    fd_man = fs_man ÷ ft_man(ループ56回);
    if(fd_man[56] == 1) then /* 桁上がりした */
        fd_exp = fd_exp + 1;
        td_man = 右シフト fd_man を 1ビット;
    endif
else
    fd_exp = fs_exp + ft_exp - 0x7f;
    fd_man = fs_man ÷ ft_man(ループ27回);
    if(fd_man[27] == 1) then /* 桁上がりした */
        fd_exp = fd_exp + 1;
        td_man = 右シフト fd_man を 1ビット;
    endif
endif
```

(a) 除算と正規化

```
q = 0;/* 商 */
r = 0;/* 剰余 */
repeat 繰り返し回数 then
    r = fs_man - ft_man;
    if(tmp ≧ 0) then
        q = q + 1;
        fs_man = r;
    endif
    fs_man = 左シフト fs_man を1ビット;
    q      = 左シフト q      を1ビット;
endrep
if(r ≠ 0) then /* Stickyがある */
    q = q or 1;
endif
```

(b) 除算のアルゴリズム

となる．仮数部の除算を引き戻し法などで計算する場合，繰り返し回数を多くすればいくらでも精度を得ることができる．しかし，今の場合は小数点以下4桁が求まればいいので5回の繰り返しになる．ただし，5回では，

0.hhhh

という結果(商)になったとき，正規化によって左シフトした場合，最下位の値がわからない．このため，必要な回数＋1回の繰り返しを行い，商を，

リスト4　平方根の処理

```
fd_sgn = 0;
if(倍精度) then
    if((fs_exp-0x3ff)のビット0が1) then
        fs_man = 左シフト fs_man を1ビット;
    endif
    fd_exp = (fs_exp-0x3ff)/2 + 0x3ff;
    fd_man = fs_man を開平計算(ループ56回);
    if(fd_man[56] == 1) then /* 桁上がりした */
        fd_exp = fd_exp + 1;
        td_man = 右シフト fd_man を 1ビット;
    endif
else
    if((fs_exp-0x7f)のビット0が1) then
        fs_man = 左シフト fs_man を1ビット;
    endif
    fd_exp = (fs_exp-0x7f)/2 + 0x7f;
    fd_man = fs_man を開平計算(ループ27回);
    if(fd_man[27] == 1) then /* 桁上がりした */
        fd_exp = fd_exp + 1;
        td_man = 右シフト fd_man を 1ビット;
    endif
endif
```

(a) 開平計算と正規化

i.iiii

jj.jjjj

という形式にしておくほうが正しい精度が求まる．除算を行った時の剰余が0でない場合は，割り算を繰り返すと0でない位の数が存在するということなので，Stickyが1になるということである．つまり仮数部の最下位を1とORしておく．

除算は，仮数部どうしを除算(正規化できるまで繰り返す)して，結果を丸めればよい．結果の符号は，ソースオペランドの符号の排他的論理和になる．結果の指数は，ソースオペランドの指数を減算したものである．アンパックとリパックを除く手順を**リスト3(a)**に示す．特殊データ型の処理や，オーバフロー，アンダフローの検出処理は省略してある．また，除算のアルゴリズムの一例を別途示す(**リスト3(b)**)．

加算の丸めと同じである．

● **平方根**

まずは10進数で平方根のアルゴリズムを検討しよう．与えられた浮動小数点を，

$a.bbbb \times 10^{c} \quad (a \neq 0)$

とする(小数点以下は4桁と仮定)．平方根の場合の結果は，

$(a.bbbb)^{(1/2)} \times 10^{(c/2)}$

となる．指数部が2で割りきれないと困るので，指数部が奇数の場合は，あらかじめ仮数部を10倍しておいて(指数部は1少なくなる)，

$ab.bbb0 \times 10^{(c-1)} \quad (a \neq 0)$

の平方根を求めればよい．仮数部の平方根が，

```
b = 0;
q = 0;
r = 0;
repeat 56 then
    s = (右シフト fs_man を55ビット) and 3;
                                    /* 下位2ビット */
    r = r + s;
    t = r - (b+1);
    if(t ≧ 0) then
        q = q + 1;
        r = t;
        b = b + 2;
    endif
    fs_man = 左シフト fs_man を2ビット;
    r      = 左シフト r      を2ビット;
    b      = 左シフト b      を1ビット;
    q      = 左シフト q      を1ビット;
endrep
if(r ≠ 0) then /* Stickyがある */
    q = q or 1;
endif
```

(b) 開平計算のアルゴリズム(倍精度)

```
b = 0;
q = 0;
r = 0;
repeat 27 then
    s = (右シフト fs_man を26ビット) and 3;
                                    /* 下位2ビット */
    r = r + s;
    t = r - (b+1);
    if(t ≧ 0) then
        q = q + 1;
        r = t;
        b = b + 2;
    endif
    fs_man = 左シフト fs_man を2ビット;
    r      = 左シフト r      を2ビット;
    b      = 左シフト b      を1ビット;
    q      = 左シフト q      を1ビット;
endrep
if(r ≠ 0) then /* Stickyがある */
    q = q or 1;
endif
```

(c) 開平計算のアルゴリズム(単精度)

d.dddddddd....

であるとすれば，結果は，

$\mathrm{d.dddd} \times 10^{(c/2)}$

となる．平方根も除算と同様に繰り返し計算で求めることができ，繰り返し回数を多くするほど多くの精度を得られる．このときの剰余に相当する値がStickyの元になる．

平方根は，仮数部を開平計算(正規化できるまで繰り返す)して，結果を丸めればよい．結果の符号は0でなければならない．結果の指数は，ソースオペランドの指数(バイアスなし)を1/2したものである．アンパックとリパックを除く手順を**リスト4**に示す．特殊データ型の処理や，オーバフロー，アンダフローの検出処理は省略してある．また，2進数での開平計算のアルゴリズムの一例を別途示す．

平方根はNewton法などで求める場合もあるが，開平計算による場合が一番高速である．

また開平計算のアルゴリズムは，除算と同様，倍精度と単精度で共通化できるがあえて二つに分けてある(**リスト4(b)**，**(c)**)．

丸め処理はこれも加算と同様である．

● 32ビット整数から単精度浮動小数点への変換

32ビット整数を単精度浮動小数点に変換する場合は，整数(絶対値)の最上位の1があるビットを，ビット26になるように左または右にシフトすればよい(**リスト5**)．この位置は，Guard，Round，Stickyを含めた正規化数になる位置である．また，変換前の32ビット整数をf_sとする．

● 倍精度浮動小数点から単精度浮動小数点への変換

倍精度浮動小数点から単精度浮動小数点への変換は簡単である．右シフトをして正規化位置までもってくればいい(**リスト6**)．指数の値(バイアスなし)の値は変わらない．ゼロの場合は処理を分ける必要があるが，ここでは言及しない．

リスト5　32ビット整数から単精度浮動小数点への変換

```
if(fs < 0) then /* 絶対値 */
    fd_sgn = 1;
    fd_man = -fs;
else
    fd_sgn = 0;
    fd_man = fs;
endif
fd_man = {0...0, fd_man} /* 32ビット長を仮定 */
clz    = fd_man をMSB(bit31)から
                          下位に連続する0の数を計数;
        /* 1の位置を検出 */
if(clz == 32) then /* ゼロ */
    fd_exp = 0;
    fd_man = 0;
else if(clz < 5) then
    fd_exp = 0x7f + (31-clz);
    fd_man = Sticky付き右シフト fd_man を
                                    (5-clz)ビット;
else /* clz ≧ 5 */
    fd_exp = 0x7f + (31-clz);
    fd_man = 左シフト fd_man を(clz-5)ビット;
endif
```

リスト6　倍精度浮動小数点から単精度浮動小数点への変換

```
fd_sgn = fs_sgn;
fd_exp = fs_exp - 0x3ff + 0x7f;
                            /* バイアス値を変えるだけ */
fd_man = Sticky付き右シフト fs_man を29ビット;
```

リスト7　32ビット整数から倍精度浮動小数点への変換

```
if(fs < 0) then /* 絶対値 */
    fd_sgn = 1;
    fd_man = -fs;
else
    fd_sgn = 0;
    fd_man = fs;
endif
fd_man = {0...0, fd_man} /* 32ビット長を仮定 */
clz    = fd_man をMSB(bit31)から
                        下位に連続する0の数を計数;
         /* 1の位置を検出 */
if(clz == 32) then /* ゼロ */
    fd_exp = 0;
    fd_man = 0;
else
    fd_exp = 0x3ff + (31-clz);

    fd_man = 左シフト fd_man を(clz+24)ビット;
endif
```

リスト8　単精度浮動小数点から倍精度浮動小数点への変換

```
fd_sgn = fs_sgn;
fd_exp = fs_exp - 0x7f + 0x3ff;
                         /* バイアス値を変えるだけ */
fd_man = 左シフト fs_man を29ビット;
```

リスト9　単精度浮動小数点から32ビット整数への変換(シフト処理)

```
cnt = fs_exp - 0x7f - 23;
if(cnt < 0) then
    fd_man = Sticky付き右シフト fs_man を -cnt ビット;
else if(cnt < 8) then
    fd_man = 左シフト fs_man を cnt ビット;
else if((cnt==8)&&fs_sgn) then
    fd_man = 左シフト fs_man を cnt ビット;
endif
else
    /* オーバフロー */
endif
```

リスト10　倍精度浮動小数点から32ビット整数への変換(シフト処理)

```
cnt = fs_exp - 0x3ff - 52;
if(cnt < -21) then
    fd_man = Sticky付き右シフト fs_man を -cnt ビット;
else if((cnt== -21)&&fs_sgn) then
    fd_man = Sticky付き右シフト fs_man を -cnt ビット;
endif
else
    /* オーバフロー */
endif
```

● 32ビット整数から倍精度浮動小数点への変換

32ビット整数を倍精度浮動小数点に変換する場合は，整数(絶対値)の最上位の1があるビットをビット55になるように左シフトすればよい(**リスト7**)．この位置は，Guard，Round，Stickyを含めた正規化数になる位置である．また，変換前の32ビット整数をf_Sとする．

● 単精度浮動小数点から倍精度浮動小数点への変換

単精度浮動小数点から倍精度浮動小数点への変換は簡単である．左シフトをして正規化位置までもってくればいい(**リスト8**)．指数の値(バイアスなし)の値は変わらない.ゼロの場合は処理を分ける必要があるが，ここでは言及しない．

● 単精度浮動小数点から32ビット整数への変換

単精度浮動小数点を32ビット整数に変換するには，指数の値(バイアスなし)が0になるように，仮数を左または右シフトすればよい(**リスト9**)．

次に丸め処理であるが，浮動小数点から整数への変換は通常切り捨て(RZ)で行われる．これはTruncate(切り捨て)変換と同じである．この他にも，整数の変換方式には，丸めモードによってCeiling(RP)，Flooring(RM)，Rounding(RN)という処理がある．

最後に，結果に符号を付ける処理は，丸められた整数は正の数なので，fs_sgnにしたがって，符号を付け直す．つまり，fs_sgnが1ならば，符号反転(0から引き算)する．

● 倍精度浮動小数点から32ビット整数への変換

倍精度浮動小数点を32ビット整数に変換するには，指数の値(バイアスなし)が0になるように，仮数を右シフトすればよい(**リスト10**)．

次に丸め処理であるが，浮動小数点から整数への変換は通常切り捨てで行われる．これはTruncate(切り捨て)変換と同じである．この他にも，整数の変換方式には，丸めモードによってCeiling，Flooring，Roundingという処理がある．

最後に，結果に符号を付ける処理は，丸められた整数は正の数なので，fs_sgnにしたがって，符号を付け直す．つまり，fs_sgnが1ならば，符号反転(0から引き算)する．

● 比較

浮動小数点の比較とは，「<(less)」，「=(equal)」，「比較不能(unordered)」という情報を確定することである．結果の表現方式は，プロセッサのアーキテクチャによってまちまちである．

普通に考えれば，浮動小数点の減算を行って，その結果を調べればいいが，浮動小数点の比較は，符号，指数，仮数を独立して比較することで，簡単に行うことができる(**リスト11**)．

● FPUのパイプライン処理

以上述べてきた浮動小数点の演算はパイプライン処

```
if(ソース1またはソース2の片方または両方がQNaNまたはSNaN) then
      less  = 0;
      equal = 0;
      unordered = 1;
else
      unordered = 0;
      equal_1 = (fs_sgn == ft_sgn);
      equal_2 = (fs_exp == ft_exp) and (fs_man == ft_man);
      tmp   = {fs_sgn, fs_exp, fs_man} - {ft_sgn, ft_exp, ft_man};
              /* 単にソース1からソース2を(整数として)減算 */
      less_1 = (fs_sgn==1)and(ft_sgn==0);
      less_2 = ((fs_sgn==1)and(ft_sgn==1)and(tmp≧0)) or
               ((fs_sgn==1)and(ft_sgn==0)and(tmp<0));
      less  = (less_1 or less_2) and (not equal);
      equal = (ソース1またはソース2の片方または両方がゼロ)? equal_2 :
               (equal_1 and equal_2);
              /* -0 と +0 は等しいことに注意 */
endif
```

リスト11 比較処理

理により操作をオーバーラップさせることができる．たとえば，MIPS R4000のFPUパイプラインは次のようなステージから構成される．

U … アンパック
S … シフト
A … 仮数部の加算
EX … 実行ステージ
M … 乗算の第1ステップ
N … 乗算の第2ステップ
D … 除算
E … 例外のテスト
R … 丸め

それぞれの命令のパイプライン構成を図6に示す．また，パイプライン処理による演算のオーバーラップ実行の例を図7に示す．

浮動小数点演算は，大まかにアンパック，演算，丸

Column 開平計算

平方根を求める方法は開平計算として知られている．開平法ともいう．図Aを参照して，1234.56789(10進数)の平方根を求める方法を示す．

(1) まず，小数点を境に2桁ずつに区切る．
(2) 最初の12に関して同じ数を掛け合わせて，それを越えない一番近い数を求めると3になる．これが平方根の最上位の値である．また，12から3×3を引くと3が残る．
(3) 差の3に次の2桁である34から1桁追加して33を作る．これを，先の掛け合わせた数の3の和(＝6)で割り，5を得る．これが平方根の次の位の値である．
(4) 33にさらに1桁追加して334を作る．これから3＋3＝6に5を追加した65に5を掛けた325を引くと9が残る．
(5) 差の9に次の2桁である56から1桁追加して95を作る．これを先の掛け合わせた数の和である70(＝65＋5)で割り，1を得る．これが平方根の次の位の値である．
(6) 95にさらに1桁追加して956を作る．これらか65＋5＝70に1を追加した701と1を掛けた701を引くと255が残る．

図A 開平計算(10進数)の例

(7) 以下，同様に繰り返す．言葉だけではわかりにくいので図を見てほしい．

以上の開平計算を2進数で行えば，状況はもっと単純である．そこに登場する数値は0か1しかない．

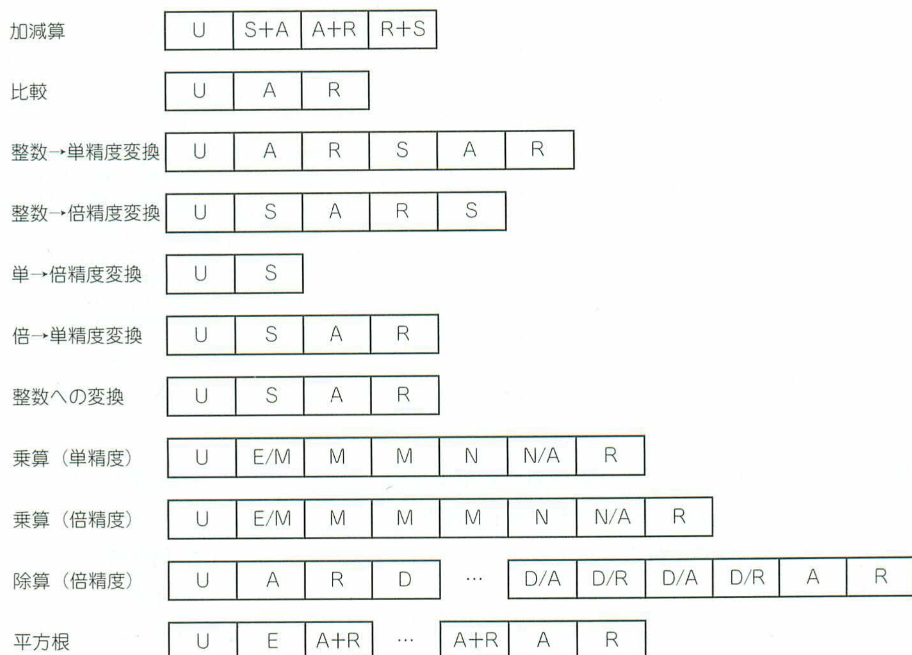


図6　R4000の浮動小数点演算のパイプライン

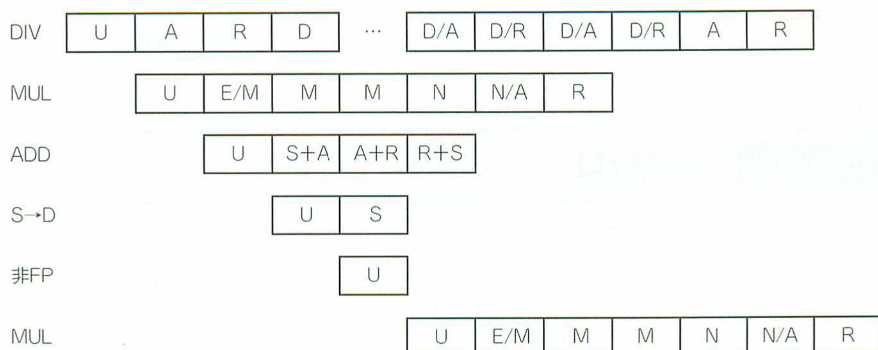


図7　パイプラインの処理例

め＆リパックに分類できる．それらは，通常，ハードウェア資源を共有しないので，スループット1クロックで浮動小数点の演算を開始できる．演算の時間は加減算で1クロック，乗算で2クロック，除算で数10クロックが普通である．もし，すべての演算時間が1クロックになれば，浮動小数点演算も整数演算と同様にパイプライン的に1クロック実行になる．これが理想だが，現実はそこまでいっていない．

なお，浮動小数点演算，とくに3次元グラフィックにおいては，除算の性能はシステム性能に依存する．たとえばグラフィックに特化したPlayStaion2のEmotionEngineは専用の除算器を用意して，除算を数クロックで行う（これができるのも単精度演算のみに割りきったことが一因ではあるが）．PlayStation2の発表当時，除算器の採用は実に画期的であった．

まとめ

FPUの概要について述べてきた．浮動小数点演算のアルゴリズムを示したが，それらはすべて整数命令で実現可能である．しかし，その処理を専用ハードウェアによって，10倍から50倍の性能を実現できる．パイプライン処理を行うことで実質的な実行時間を短縮することもできる．現在，浮動小数点の演算の処理時間は，整数演算の処理時間とますます変わらなくなってきている．この進歩が，3Dグラフィックの応用分野を飛躍的に広げたのは疑いようがない．また，MPUの開発においても整数性能は飽和してきているが，浮動小数点性能は確実に向上している．その基本となるFPUの原理を知っておくことは必須の教養であろう．

Appendix 6

演算回路をいかに高速に処理するか

高速演算器の実際

高速な整数演算や浮動小数点演算を行うためには，高速な演算器が必要である．ここでは，高速演算器の実際について説明する．CPUやFPUを設計する場合(は，あまりないと思うが)の参考になれば幸いである．

● 加算器の実際

加算器においては，まず1ビットの加算を考える．2進数では，

0 + 0 = 0　桁上がり　0
0 + 1 = 1　桁上がり　0
1 + 0 = 1　桁上がり　0
1 + 1 = 0　桁上がり　1

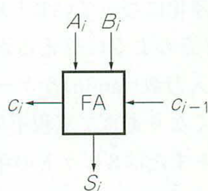
を実現できればいいので

$S = A$ xor B　(和)
$C = A$ and B　(桁上がり)

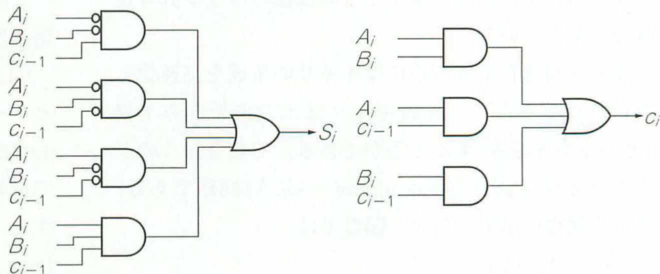
で示すことができる．これが半加算器(Half Adder)の論理である．しかし，半加算器は下位からの桁上がり(キャリ)を足し込めないので現実的でない．実際には図A(a)に示す真理値表を実現する論理が必要である．それが全加算器(Full Adder)で，図A(b)または図A(c)で示される．1ビットの全加算器があれば，それを直列に組み合わせて任意のビット数の加算器を構成することができる．図Bの加算器はキャリが1ビットずつ，さざ波のように伝播していくので，リプル(さざ波)キャリ加算器(Ripple Carry Adder)と呼ばれる．

リプルキャリ加算器の欠点は，下位のキャリが決定しないと，そのビットの和が求まらない点である．図A(b)から1ビットのキャリを生成するためには2段のゲートを通らなければならない．一つのゲート遅延を

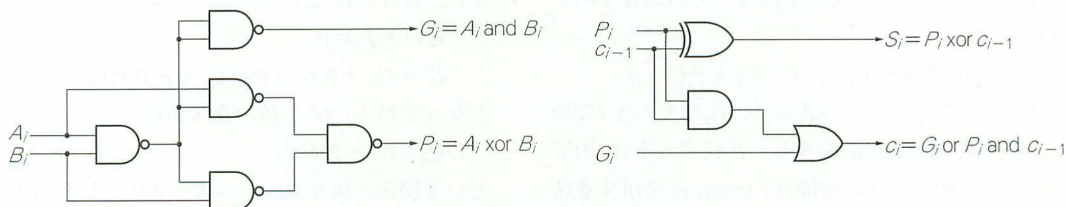
A_i	B_i	c_{i-1}	S_i	c_i
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



(a) 1ビット全加算器の真理値表

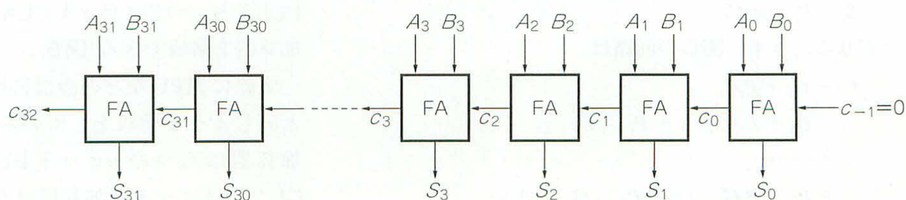


(b) 1ビット全加算器の構成(その1)

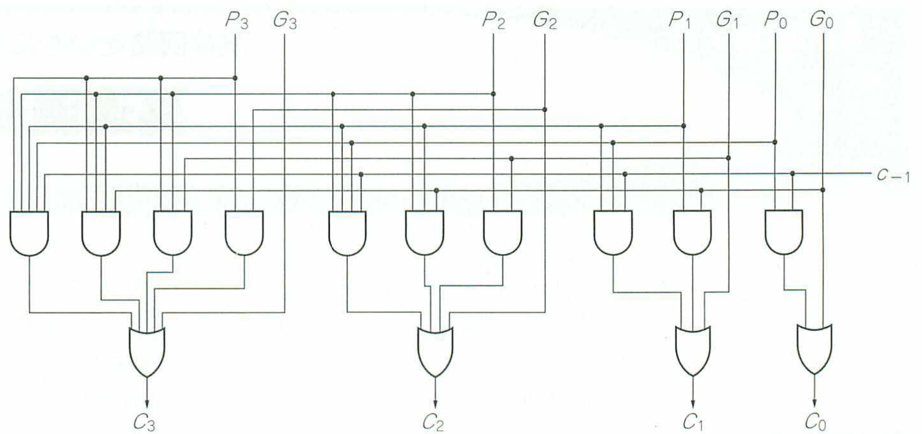


(c) 1ビット全加算器の構成(その2)

図A　1ビット加算器



図B　32ビットRCA
(Ripple Carry Adder)



図C
CLA(Carry Look Ahead)**回路**

Δとすれば，nビットのリプルキャリ加算器の計算時間は$2n\Delta$となる．**図A(c)**では1ビットの和を計算するのに4段のゲートを通過しているように見えるが，現実には**図A(b)**のように3段(キャリの反転に1段必要)で構成可能である．これを見ると和の計算のほうが律速段階に思えるが，キャリが伝播するうちに和は計算できるので，和を3段で計算する場合には$(2n\Delta+1)$でnビットの計算が終わる．+1は誤差ともみなせるので，実際にはキャリの伝播のほうが計算速度を支配している．

加算を高速化するためにはキャリの生成を高速化しなければならない．実はキャリは入力の値から予測(というか先読み)することができる．それを行うのがキャリ先見(Carry Look-ahead：CLA)回路である．それを**図C**に示す．なお，**図C**では，

$$G_i = A_i \text{ xor } B_i$$

$$P_i = A_i \text{ and } B_i$$

である．このG_iとP_iを使用すれば，その加算でのキャリC_iは，

$$C_i = G_i \text{ or } P_i \text{ and } C_{i-1} \quad (= G_i + P_i C_{i-1})$$

で示すことができる．この式より，G_iはビットiでのキャリの生成(Generate)を示し，P_iはビットiでの下位ビットからのキャリの伝播(Propagate)を示す意味があることが分かる．ついでにいえば，A_iとB_iの和S_iはP_iと下位からのキャリC_{i-1}を用いて，

$$S_i = P_i \text{ xor } C_{i-1}$$

と表せる．本来，**図C**の回路は，

$$\begin{aligned} C_i &= G_i + P_i C_{i-1} \\ &= G_i + P_i(G_{i-1} + P_{i-1}C_{i-2}) \\ &= \cdots\cdots \\ &= G_i + P_i G_{i-1} + P_i P_{i-1} G_{i-2} + \cdots \\ &\quad + P_i P_{i-1} \cdots\cdots P_1 G_0 + P_i P_{i-1} \cdots\cdots P_0 C_{-1} \end{aligned}$$

で$i=4$の場合をゲートで表したものである．G_iやP_iを生成するのにゲート1段，キャリの先見に2段(**図C**)，和に3段で6Δの時間で計算可能である．この時間は加算するビット数にかかわらず一定である．この結果，CLAを使用すれば，

$$(2n\Delta)/(6\Delta) = n/3$$

倍の高速化が実現できる．一般には(nが大きい場合)，

$$n/\log(n)$$

倍程度の高速化になっているという．

CLAは万能のように思えるが，ビット数が増えるとゲートの入力数(fan-in)やゲートからの出力数(fan-out)が大きくなり過ぎて実現不可能になる．現実としては4ビットまたは8ビットの単位に切り分けて計算する．この方式をブロックキャリ先見(Block Carry Look Ahead：BCLA)という．**図D**が4ビットのブロックキャリ先見回路で，ここではC_3を計算する代わりに，新しいG^*とP^*を生成している．ここで，

$$P^* = P_3 P_2 P_1 P_0$$

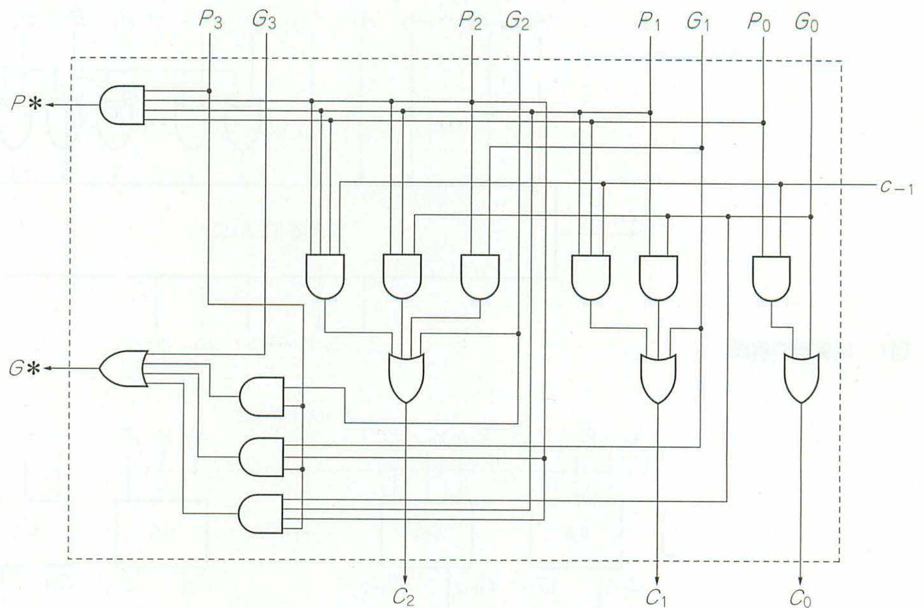
$$G^* = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$

となっており，G^*とP^*の意味から，

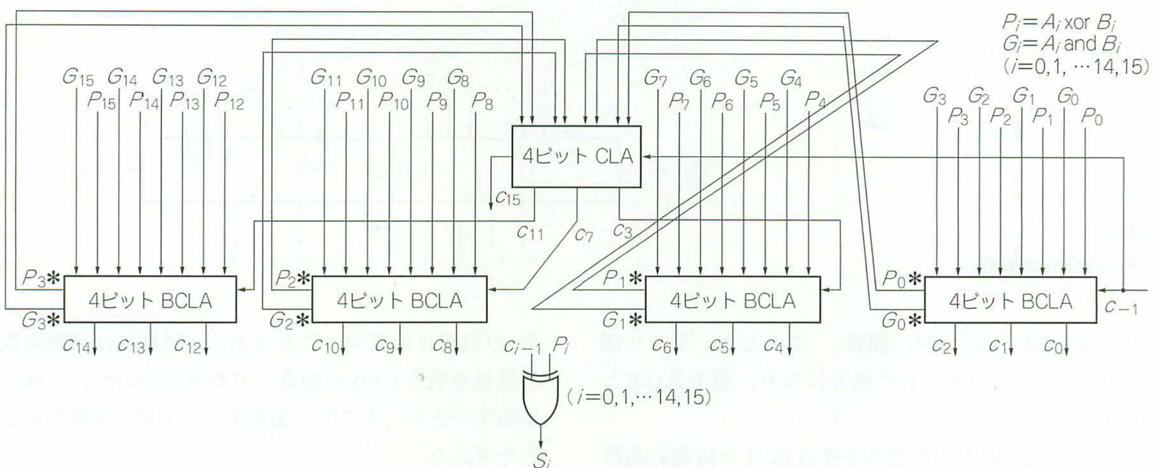
$$C_3 = G^* + P^* C_{-1}$$

という関係が成り立つ．つまり4ビットのBCLAを1ビットの生成(G^*)，伝播(P^*)，下位からのキャリ(C_{-1})とみなせる．これに注目し，四つの4ビットBCLAと一つの4ビットCLAを使用して16ビットの加算器を構成できる(**図E**)．

実際にMPUなどの設計に用いられる加算器は**図E**と同じような構成をしている．ただし，32ビットの加算器は八つの4ビットBCLAと一つの8ビットCLA，64ビットの加算器は八つの8ビットBCLAと



図D
4ビットBCLA（Block Carry Look Ahead）回路



図E　BCLAとCLAを組み合わせた加算器

一つの8ビットCLAで構成される．CLAはBCLAで代用される場合もある．

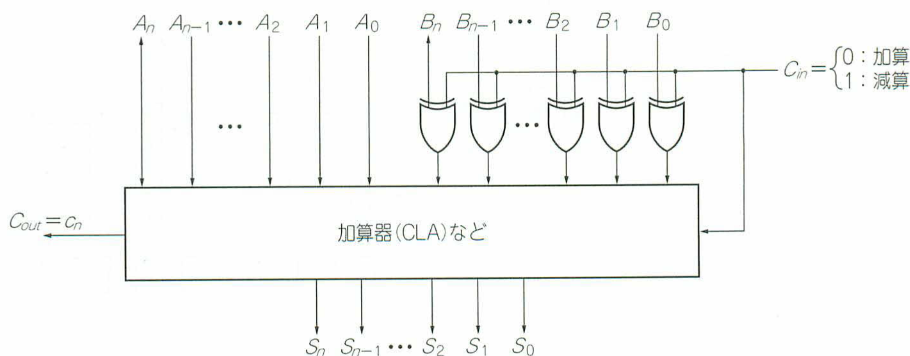
なお，ここで紹介した加算機はキャリを伝播して計算する方式を採るので一般的にキャリ伝播加算機(Carry Propagation Adder：CPA)と呼ばれる．加算機の構造に言及したい場合は，同じCPAでもRCA(Ripple Carry Adder)とかCLAとかBCLAと呼んで区別する．

● 減算器の実際

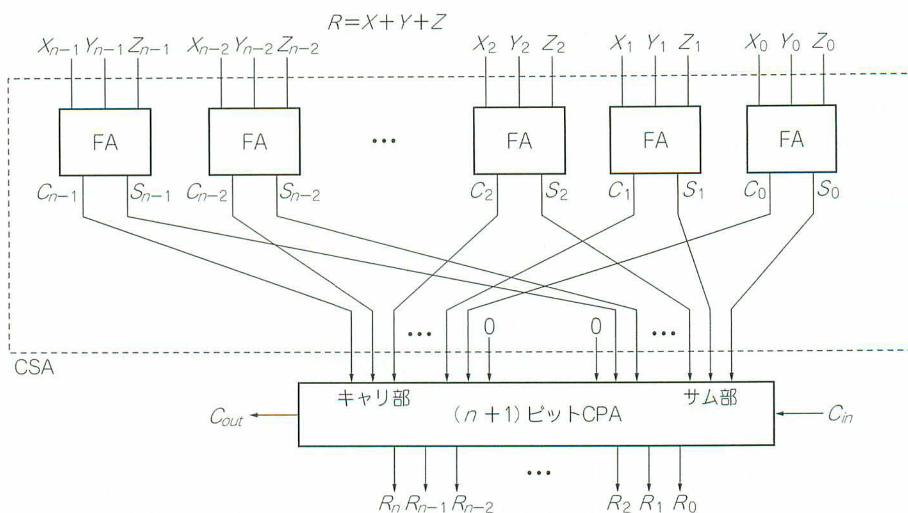
減算器は加算器の第2入力を負の数に変換して入力することで実現する．つまり，第2入力の2の補数を取って加算する．2の補数はビットを反転して1を加算することで実現できるので，この1を最下位からのキャリ入力にしてやれば，第2入力は反転するだけでよい．これは1と排他的論理和を取ることで計算できる．結局，加減算器の構成は図Fのようになる．

● 多入力の加算器

キャリの伝播が加算時間の律速段階になることはすでに述べた．それならば逆の発想で，キャリを伝播させずに加算を行うことが考えられる．これは特に3入力の加算で多用される．全加算器(FA)において，キャリ入力を第3の入力に見立てればよい．この三つの入力を加算しておいて，キャリの集まりと和(サム)の集まりを別個に計算しておく．これをキャリ保存加算器(Carry Save Adder：CSA)と呼ぶ．つまり，まずCSAでキャリ部とサム部を計算しておき，最後にそ



図F　減算器の構成



図G
キャリ保存加算器

れらをCPAで足し込む(**図G**)．このとき，キャリ部は1ビット左シフトされた格好になり，最下位は0となる．

もし，三つの入力のどれかを反転(1と排他的論理和)してキャリ部の最下位に1を入れておくと減算を行うことになる．

CSAが一つだけではありがたみも少ないが，**図H**のようにCSAを木構造に結合すると高速な多入力の加算器を構成できる．多入力の加算を行う場合は3入力単位に分けてCSAに入力する方法が好まれる．**図H**も右側から3入力をなるべく組み合わせた構成になっている．

この木構造をWallaceツリーと呼ぶ．CSAの組み合わせ論理をVLSI上に集積する場合は，配線の規則性が問題となる．Wallaceツリーはその要求に応える構造になっている．Wallaceツリーは乗算器において部分積を足し合わせる場合には特に多用される．

● 乗算器の実際

乗算の原理は掛けられる数(被乗数)を掛ける数(乗数)の数だけ加算することである．単純には乗数の数だけ被乗数をCSAで加算してやればいいが，これでは効率が悪い．そこで，乗数をビット列に分解することを考える．

$X \times Y$

を計算する場合，

$Y = Y_{n-1}Y_{n-2}\cdots Y_2Y_1Y_0$　(Y_iは0か1)

とすると，

$$Y = (Y_{n-1} << (n-1)) + (Y_{n-2} << (n-2)) + \cdots\cdots + (Y_2 << 2) + (Y_1 << 1) + Y_0$$

とn個の和に分解できる(<<は左シフトを表す)．このとき積も，

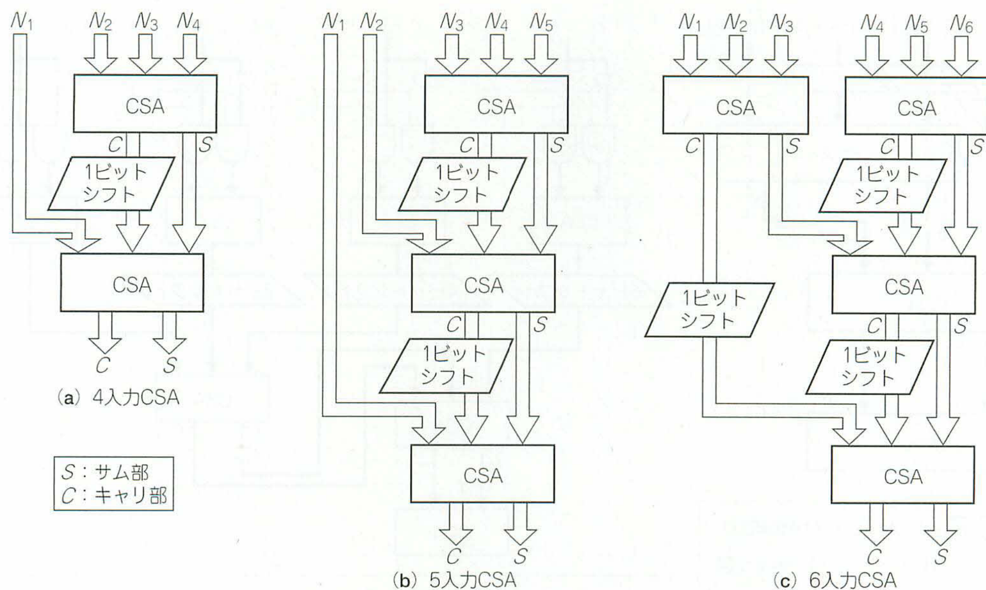
$$X \cdot Y = (X \cdot Y_{n-1} << (n-1)) + (X \cdot Y_{n-2} << (n-2)) + \cdots\cdots + (X \cdot Y_2 << 2) + (X \cdot Y_1 << 1) + X \cdot Y_0$$

とn個の和になる．これをCSAで加算する．たとえば，$n = 6$の場合，

$N_1 = X \cdot Y_5 << 5$

$N_2 = X \cdot Y_4 << 4$

$N_3 = X \cdot Y_3 << 3$



図H　多入力CSAの構成例

$N_4 = X \cdot Y_2 << 2$

$N_5 = X \cdot Y_1 << 1$

$N_6 = X \cdot Y_0$

として，**図H(c)**の6入力CSAに入力してキャリ部とサム部を得て，それらをCPAで加算すれば積が求まる．

今は乗数にかんして1ビットずつ処理を行ったが，これは1ビットずつ部分積を計算して足し合わせた．これを2ビットずつ処理すれば，部分積の加算回数が半分になる．Yを2ビットずつ処理するとき，その組み合わせには次の4種類があり，その値に応じて0，X，$2X$，$3X(= X + 2X)$を選択してCSAで加算していけばいい．

$\{Y_{i+1},\ Y_i\}$ = 00 … 0を加える
01 … Xを加える
10 … $2X$(Xの1ビット左シフト)を加える
11 … Xと$2X$を同時に加える

この方式の乗算器($n = 8$の場合)を**図I(a)**に示す．この方式の欠点は，**図I(b)**に示すように，$3X$を作るために余分な加算が必要なことである．被乗数を加える際にシフトするだけで目的の部分積が得られることがゲート数を削減するためにも望ましい．これを満たす方法としてBoothのアルゴリズムがある．

Boothのアルゴリズムは，処理単位のビットの他にさらに下位の1ビットを見て足し込む部分積を決定する．具体的には乗数のビットを3ビットずつ見ていくが，ビットを走査する開始位置を2ビットずつずらしていく．このとき部分積は，

$\{Y_{i+1},\ Y_i,\ Y_{i-1}\}$ = 000…0を加える
001…$+X$を加える
010…$+X$を加える
011…$+2X$を加える
100…$-2X$を加える
101…$-X$を加える
110…$-X$を加える
111…0を加える

となる．これなら被乗数のシフトと符号反転(減算)だけで実現できる．ただし，Y_{-1}は0である．

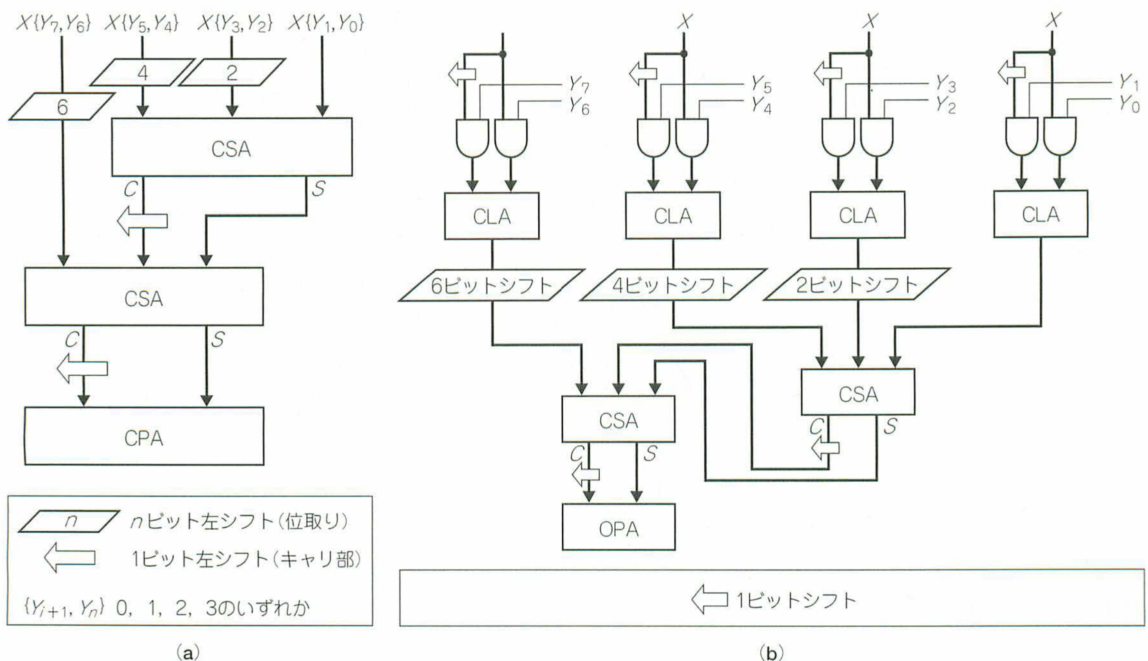
具体例を示そう．X = 0x34，Y = 0x56としてXY = 0x1178を手計算してみる．いま8ビット×8ビットの乗算を規定しているので，結果は16ビットとなる．まず**図J(a)**のように$+X$と$+2X$を計算する．2倍する場合は1ビット左シフトするだけである．

次に，$-X$と$-2X$を計算する．これは2の補数なので0と1を反転して1を加えればいい(**図J(b)**)．

次にYのビットを走査する．3ビットずつ見ていくが，ビットを走査する開始位置を2ビットずつずらしていく．この例では最後にビットが足りなくなるので最下位に0を追加すると，

$-2X + 2X + X + X$

を加算すればよいことになる(**図J(c)**)．以上を**図J(d)**のように足し合わせると0x1178という結果が得られ



(a)　　(b)

図I　2ビット単位に処理する乗算器の例

```
 +X=0000 0000 0011 0100
+2X=0000 0000 0110 1000  ←  1ビット左シフト
```

(a) +Xと+2Xの計算

```
 −X =1111 1111 1100 1011 + 1
    =1111 1111 1100 1100
−2X =1111 1111 1001 0111 + 1
    =1111 1111 1001 1000
```

2の補数なので0と1を反転して1を加える

(b) −Xと−2Xの計算

```
Y =0101 0110 0 ← ゼロを加える
          10⓪ … −2X & 0ビット左シフト
        011   … +2X & 2ビット左シフト
      010     … +X  & 4ビット左シフト
    010       … +X  & 6ビット左シフト
```

(c) ビットの走査

```
    1111 1111 1001 1000
    0000 0000 0110 1000 << 2ビット左シフト
    0000 0000 0011 0100 << 4ビット左シフト
+)  0000 0000 0011 0100 << 6ビット左シフト
-----------------------
       を計算すると…

    1111 1111 1001 1000 ┐
    0000 0001 1010 00   ┘
    0000 0011 0100
+)  0000 1101 00
-----------------------
    0000 0001 0011 1000 ← 上二つを加算
    0000 0011 0100
+)  0000 1101 00
-----------------------
    0000 0100 0111 1000 ← 上二つを加算
    0000 1101 00
-----------------------
    0001 0001 0111 1000
              ↓
0x  1    1    7    8
```

(d) 加算処理

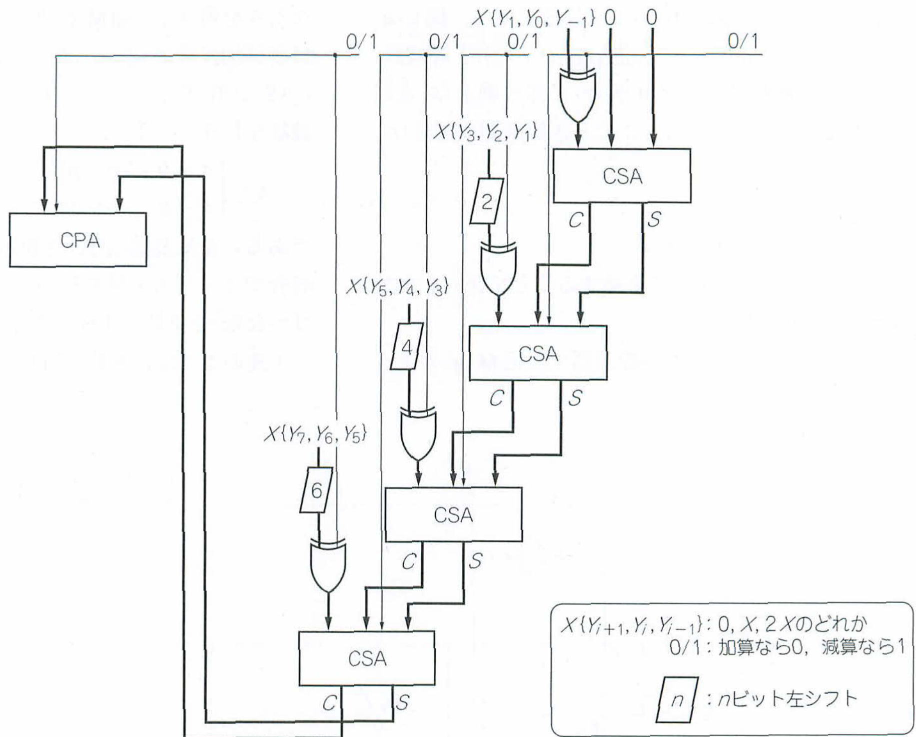
図J　X=0x34，Y=0x56としてXY=0x1178

る．これをもとにして，8ビットの乗算器をCSAを用いて構成すると**図K**のようになる．一番右上の，

$0+0+X\{Y_1,\ Y_0,\ Y_{-1}\}$

を計算するCSAは冗長である．0を加算しても同じ値になるからだ．しかし対称性がいいのでそのままにしておく．

乗算器の基本は以上のようなものである．乗算器は，一言でいえば，並行に計算可能な部分積を足し合わせるだけであるが，CSAのつなぎ方に種々のバリエーションがある．そして，それが乗算器を特徴づけているのだが，ここではこれ以上言及しない．回路に落とす場合には正方形になるような配置が好まれる．なお，実際の乗算器ではシフト処理は配線の付け替えで行うので，一般にいうシフタは不用である．



図K　Boothアルゴリズムを利用した8×8ビット乗算器

乗算器はシステムの性能を決定づけるため，回路構成や配置法がいろいろ研究されている．そのケースとして54ビット×54ビットの無符号乗算器の研究が多い．これは浮動小数点の乗算を高速に計算する需要がもっとも多いからである(精度はGurardとStickyの2ビットを想定している)．

● 除算器の実際

除算器は，乗算器とは異なり被除数から除数の減算で構成される．しかし，部分商といったものを並列に計算することはできず，必ず一つ前の結果に依存して次の処理を決定しなければならない．事実，除算器というものは存在するが，これが決定版というものはない．それだけ多くの研究があるのも事実であるが，ここでは簡単な除算器の例(引き戻し法と引き放し法をそれぞれ一つ)を述べるに留める．

なお，ここで紹介する除算器では，被除数，除数ともに正の数であることと仮定している．本来，浮動小数点の仮数部は符号と小数部の絶対値からなるので，これで十分である．

除算においては，被除数から除数が引ける(ボロウが出ない)場合に1が立ち，逆の場合に0が立つ．このときの差が部分剰余として新たな被除数になる．引き戻し法とは，引けない(ボロウが出る)場合には部分剰余に除数を加えて元の値に戻す方法である．本Appendixで除算のアルゴリズムとして上述してあるのが引き戻し法である．一方，引ける/引けないに無関係に必ず差を部分剰余とするのが引き放し法である．引き放し法で商として1が立つ(引ける)場合は，次の回は加算を行い，商として0が立つ(引けない)場合は次の回に減算を行う．引き戻し法のアルゴリズムに倣って手順を示すと**リストA**のようになる．

リストA　引き戻し法のアルゴリズムによる減算処理

```
q = 0;/* 商 */
r = 0;/* 剰余 */
b = 0;/* ボロウ */
repeat 繰り返し回数 then
    if(b==0)then
        r = fs_man - ft_man;
        b = 減算のボロウ;
    else
        r = fs_man + ft_man;
        b = 加算のキャリの反転;
    endif
    q = q or NOT(b);
    fs_man = r;
    fs_man = 左シフト fs_man を1ビット;
    q      = 左シフト q      を1ビット;
endrep
if(b!=0)then
    r = r + ft_man;
endif
```

さて，引き戻し法による除算器では**図L(a)**のよう

なセルを用意し，図L(b)のように構成する．図L(a)は制御減算器(Controlled Subtractor：CS)と呼ばれ，二つの数の減算によるボロウ(P)と部分剰余(S)を組み合わせ回路で生成する．このとき制御抑止信号(D)によって，Sの値は，

$$S=\begin{cases}A & (D=1)\\ A-B & (D=0)\end{cases}$$

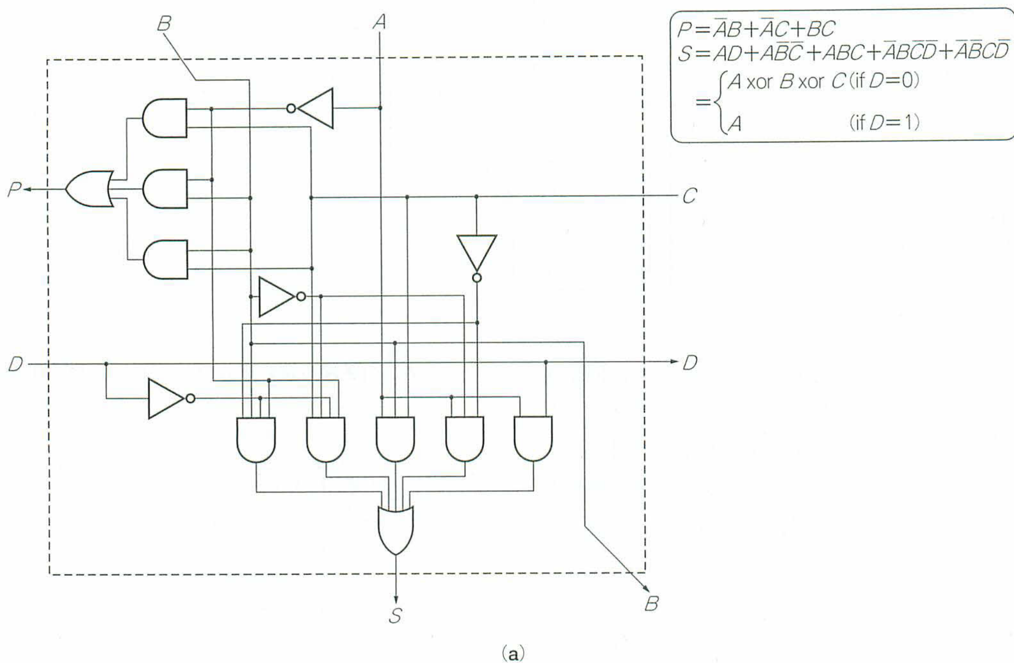
となる．つまり，PをDに直結することで正しい部分剰余が得られる．

また，引き放し法による除算器では図M(a)のようなセルを用意し，図M(b)のように構成する．図M(a)は制御加減算器(Controlled Adder or Subtractor：CAS)と呼ばれ，ボロウ(P)の値によって加算または減算を行う．つまり，

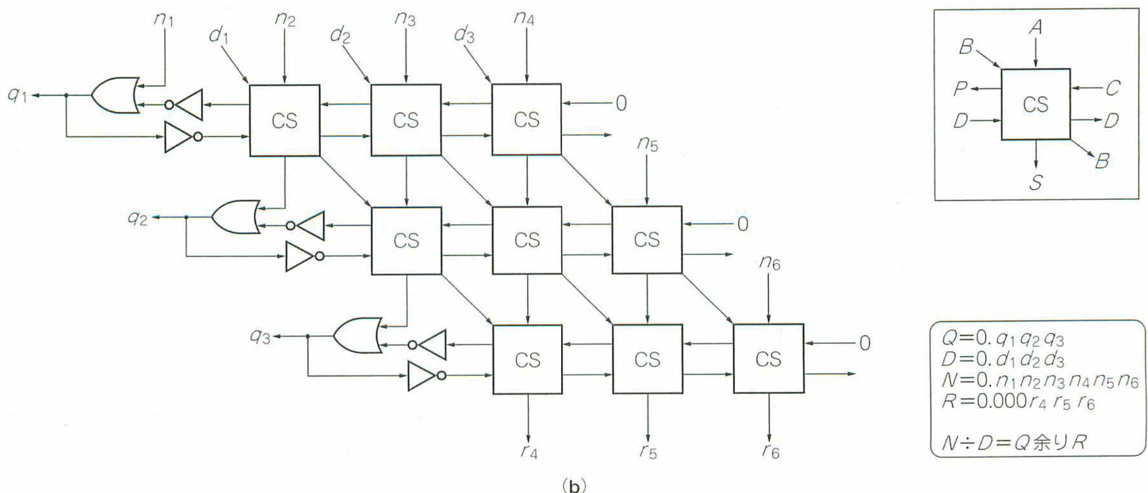
$$S=\begin{cases}A+B & (P=0)\\ A-B & (P=1)\end{cases}$$

である．全加算器(FA)を使用するため，図M(b)の各行でキャリの先見を行い，次の行へのボロウ(キャリの反転)を早期に生成することも可能である．

上述のように，乗算と同じく組み合わせ回路で除算



(a)



(b)

図L　**制御加算器**(CS)

器を構成することも可能であるが，現実のMPUに適用されている例は稀である．除算の出現頻度に比して回路規模が大きくなり過ぎるのがその理由であろう．多くのMPUでは，除算のアルゴリズムで示した繰り返し計算によって除算を実現している．

● バレルシフタの概念

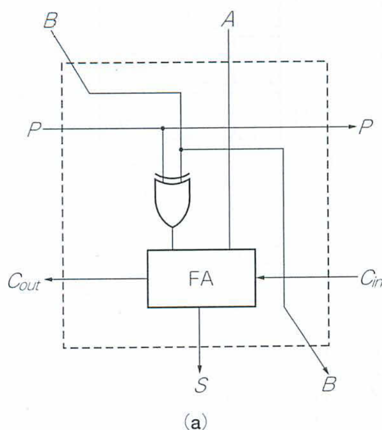
演算器の中で忘れていけないものにシフタがある．昔はフリップフロップで構成されたシフトレジスタを利用してシフタを実現していた．この場合，1クロックに1ビットしかシフトできないので，nビットシフトをするためにはnクロックの時間がかかる．これをシフト量にかかわらず，1クロックでシフト操作を実現するのがバレルシフタである．最近は，シフタと言えばバレルシフタのことを指す．

バレルシフタの構造は非常に単純であり，あえて解説する必要もないと思うのだが，一応概念だけ示しておこう．

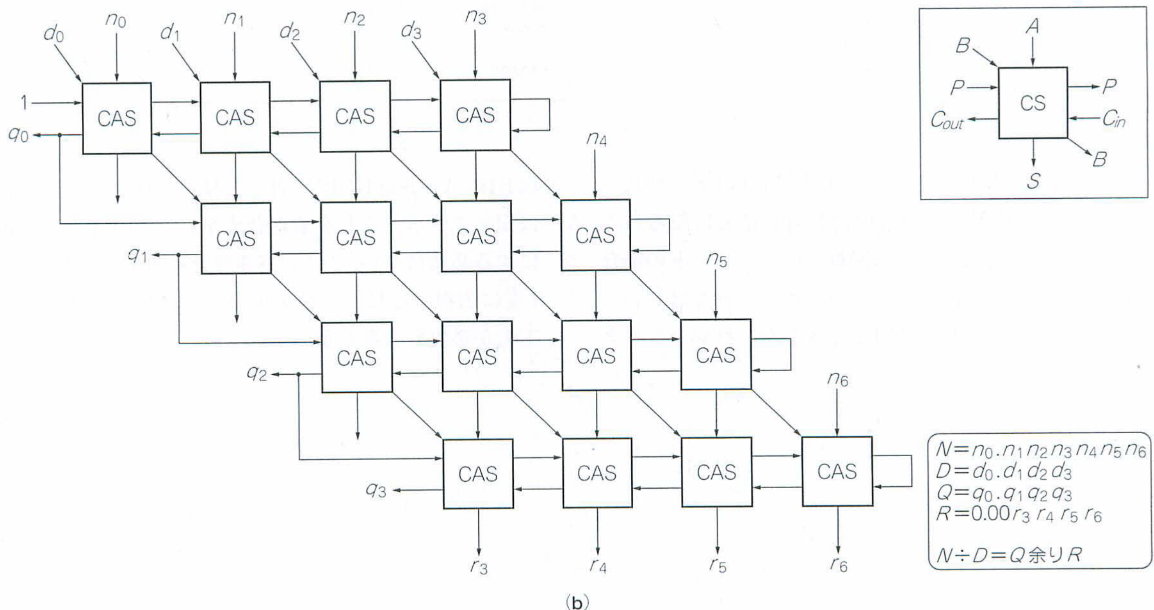
バレルシフタとはセレクタである…という一言では説明にならないので，もう少し説明しよう．簡単のために4ビットのバレルシフタを考える．**図N**に示すように，4ビットのデータを格納するレジスタ(ここでは右側をSFT0，左側をSFT1と呼ぶ)を2組用意し，シフトするデータを次のように格納する．

1) 右論理シフト：SFT0にシフトするデータ，SFT1にオール0
2) 右算術シフト：SFT0にシフトするデータ，SFT1にシフトするデータの符号ビットをデータ長だけコピーしたデータ．つまり，オール1かオール0である
3) 左(論理/算術)シフト：SFT0にオール0，SFT1にシフトするデータ
4) ローテート：SFT0とSFT1の両方にシフトするデータ

このようにシフトするデータを設定した後，SFT1とSFT0を連続する8ビット(データ長の2倍)のレジスタとみなし，右シフトなら右側(最下位ビット)のシフト量の位置からデータ長(4ビット)だけ取り出す．左シフトなら左側(最上位ビット)のシフト量の位置からデータ長(4ビット)だけを取り出す．左シフトの場

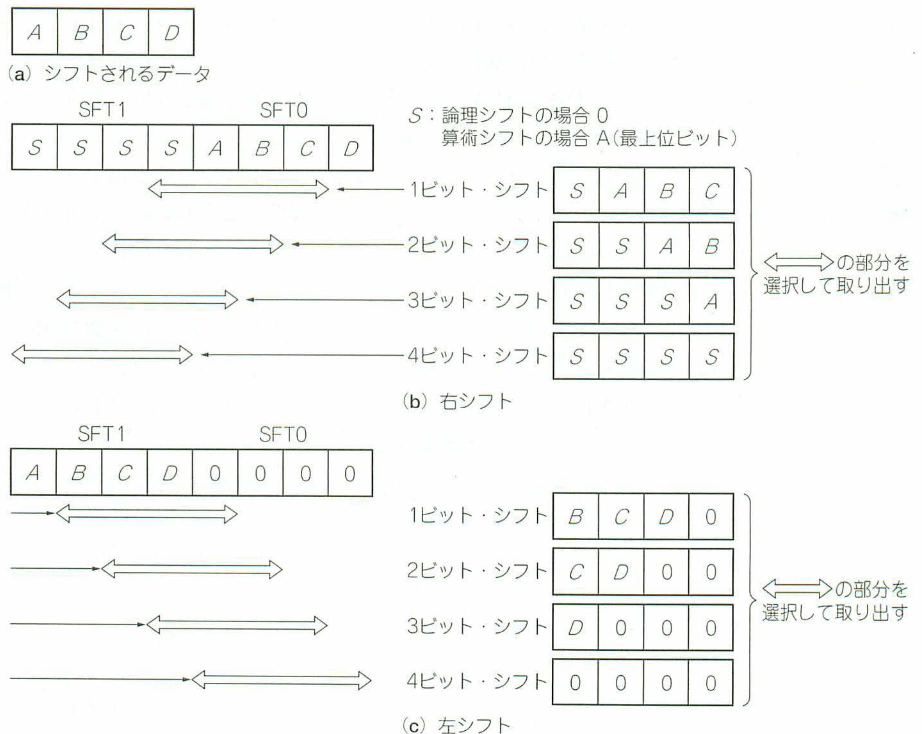


(a)



(b)

図M　制御減算器(CAS)



図N　バレルシフタの概念

リストB　バレルシフタの記述例(Verilog-HDL)

```
reg [3:0] in;   //  シフトされるデータ
wire[3:0] sft;  //  シフトした結果
reg [3:0] lsft; //  左シフトの結果
reg [3:0] rsft; //  右シフトの結果
reg [1:0] cnt;  //  シフト量
reg       ls;   //  シフト方向：左→1，右→0

always@ (in or cnt)
case (cnt)
2'd0: lsft = in;
2'd1: lsft = {in[2:0], 1'd0};
2'd2: lsft = {in[1:0], 2'd0};
2'd3: lsft = {in[0], 3'd0};
default: lsft = 4'd0;
endcase

always@ (in or cnt)
case (cnt)
2'd0: rsft = in;
2'd1: rsft = {1{in[3]}, in[2:0]};
2'd2: rsft = {2{in[3]}, in[1:0]};
2'd3: rsft = {3{in[3]}, in[0]};
default: rsft = 4{in[3]};
endcase

assign sft = (ls)? lsft : rsft;
```

合で，最下位ビットを基準にする場合は(データ長−シフト量)の位置からの選択でも同じ結果になる．これでシフトが実現できる．**図N**にはローテートの場合を図示してないが同じことなので考えてみてほしい．

…ともっともらしく説明してきたが，バレルシフタはHDL(Verilog-HDL)で書くと**リストB**のように簡単に記述できる．これを論理合成すればその構造など気にする必要はない．ただ，高速性を考慮するとなると，1度に選択するビット数を少なくするなど，いろいろ工夫が必要ではあるが….

第10章

Javaアプリケーションを高速に実行するための Javaプロセッサの特徴と実際

Javaで書かれたアプリケーションが使われる身近な例としては携帯電話があるだろう．しかし実際にはそこにJavaプロセッサが使われている例は少なく，通常はプロセッサ上で走るJava仮想マシンか，一部の実行を高速に行うためのJavaアクセラレータが内蔵されている程度である．ここではJavaプロセッサとJavaアクセラレータについて解説する．

最近，Javaというコンピュータ言語が普及しつつある．書店の店頭を見るとJava関連の書物がいかに多いことか．Javaの利点は，Javaで作成したソフトウェア(Javaアプレット)がOSやハードウェアに依存せず，ほとんどすべてのコンピュータで動作することである．最近では携帯電話でさえ動作する．この普遍性はOSに組み込まれたJava仮想マシン(JavaVM)というソフトウェアによって実現される．C言語などもほとんどのOSで動作するが，Javaの最大の特徴は(基本的に)インタプリタであるということである．アプレットを作るとすぐに動作確認ができ，修正も容易である．

JavaVMは，IEやネットスケープなどのWebブラウザでも利用できるので，インターネットを通じて共通なJavaアプレット(ゲームなど)を実行できる．Javaの普及にはインターネットの普及が一役買っている．インターネットも今後ますます発展が予想され，Javaもさらなる発展が期待できる．

このように世間がJava一色になりつつある(というのは言い過ぎだが)状況で，Javaを高速に実行する要求が発生した．それに対する回答の一つがJavaプロセッサである．つまり，JavaVMのハードウェア化である．本章ではJavaプロセッサの特徴について解説する．

1 SunのJavaプロセッサ

● Javaプロセッサとは何か

Javaプロセッサとは Javaを高速に実行するためのプロセッサである．おもにJavaのバイトコードの解釈や実行処理をハードウェア化したプロセッサのことを指す．Javaチップともいうが，JavaChipはSun Microelectronicsの登録商標らしく，Javaアクセラレータ，またはJavaプロセッサという呼称が一般的に使用される．

最近のJavaプロセッサの多くは，Javaを高速に実行できることを目標にしているが，Java専用ではないことに注意が必要である．実はバイトコード以外も実行できる．Javaプロセッサ自体の発表は各社から数多くされているが，思ったほど性能が出ないせいなのか，ライセンス料の問題なのか，実用になっているものがほとんどない．Javaプロセッサ単体としてはコストパフォーマンスが悪いので，汎用CPUの一拡張機能としてJavaの高速実行支援機能を実装する傾向にある．

ここでは各社のJavaプロセッサの特徴と状況を順に見ていこう．

● picoJava，microJava，UltraJava

1996年2月2日，Sun Microsystemsの子会社であるSun Microelectronics(以下，単にSunと呼ぶ)はJavaに最適化されたマイクロプロセッサファミリの開発を表明した(**表1**)．それが，picoJava，microJava，

表1
Sun MicroelectronicsのJavaプロセッサ

製品名	picoJava	microJava	UltraJava
概要	IPコア	PicoJavaに周辺機能を内蔵	VIS命令セットを備え，グラフィック機能を強化
用途	携帯電話，家電製品	通信機器，ゲーム機，PDA	インターネット端末，3Dマルチメディア機器
出荷予定(当初)	1996年2Q	1997年1Q	1997年4Q
価格	25ドル程度のライセンス料	25～50ドル	100ドル以上

UltraJavaである．

picoJavaはJavaの処理系のコアとなる部分であり，microJavaはpicoJavaを含みDRAMコントローラなどを加えたマイクロプロセッサである．主としてローエンドの組み込み制御分野をターゲットとし，25～50ドルでの提供を目標としている．

UltraJavaはmicroJavaのハイエンド向けである．UltraJavaはpicoJavaコアを含むか否かは明言されていない．発表時点では将来的な製品というイメージしかなかった(のちにMAJCとして再発表される)．こちらは100ドル程度の価格帯を目標としている．

Sunによれば，Javaプロセッサの性能は通常のJavaインタプリタの約12倍，Pentium(66MHz)でJITコンパイラを使用した場合と比べても3～5倍の性能を発揮できるとしている．

Sunは，その後Javaが大幅に普及するとして，2000年には150億ドル規模の市場になると予測していた．そこにJavaの高速化の要求が生まれ，Javaプロセッサのライセンス料は相当なものになると予測していたに違いない．

Sunは，基本的に，picoJavaの論理設計を行い，ライセンスを結んだ半導体メーカーにIP(Intelectual Property：知的財産．具体的には論理設計情報)として供給する．各半導体メーカーは，各社のMPUのコプロセッサとしてpicoJavaを実装して販売する．その意味で，Javaプロセッサとしてユーザーに供給されるのはmicroJavaということになる．picoJava自体はSoC(System-On-Chip)を形成するマクロという位置付けである．

1997年4月，東芝はmicroJavaの仕様をSunと合同で開発していくことを真っ先に表明した．プレスリリースによると，microJavaの製品化はSunによって1998年中頃に行われるとなっていた．東芝からJavaプロセッサが提供されるという記述はなく，この開発の東芝の役割はいまひとつ不明である．

1998年3月3日付けのEETimesでは，現時点でpicoJavaが動作するチップはないと報道され，Sunの苦境が見てとれる．ただし，NEC，富士通，LG Semiconductor，Rockwell，Siemensの5社がpicoJavaのライセンスを受けていることがわかる．NECとLGはすぐさまJavaプロセッサを出荷予定と報道されたが，その形跡はない．同記事は，IBMがpicoJavaとPowerPCをASICのライブラリとして提供することを伝えるものだったが，こちらも消息不明である．

IBMは，同時期picoJavaとは別個に，VLIW型のJavaプロセッサを開発していた．これはPowerPCの命令セットをもち，JavaのバイトコードをVLIWに変換して実行するものだった．しかし，これも研究所レベルで終わっている．

事実，SunのJavaプロセッサは世間から忘れ去られつつあった．これは，picoJavaが予定していたとおりの性能を引き出せないことが原因である．そこで，SunはpicoJavaをpicoJava-Ⅱにバージョンアップしてライセンシに供与を始めた．

2000年4月5日，富士通はpicoJava-Ⅱコアのチップ「MB86799」を開発したと発表した．同チップはpicoJava-Ⅱとしては世界初のチップ化で，実際に機器に組み込む際には，MB86799をコアとしたASICを出荷するとしている．

富士通に遅れること約2年，2001年11月15日，NECが自社のV850のコプロセッサとしてpicoJava-Ⅱを1チップに実装した「V850＋J」の開発を完了したと発表した．プレスリリースでは詳細は不明だが，同チップはSONYやNECの携帯電話のJavaアプリケーションを高速に実行するために開発されたといわれている．

2000年までJavaプロセッサが登場しなかったのは，各半導体メーカーはpicoJavaをライセンスしてみたものの，その使い道に苦慮していたものと思われる．早い話，商談がなかった．それが携帯電話という具体的な応用分野を得て製品化に至ったのであろう．

● picoJavaの特徴

picoJavaコアの最大の特徴はスタックマシンである

ことだ．これはJavaのバイトコードの特徴でもあるが，スタックにデータを積み，スタック上で演算を行い，そしてスタックから結果を取り出すというアーキテクチャを採用している(**図1**)．

Javaのバイトコードは228命令が定義されているが，picoJavaはその約95％をハードウェアで実行する．その昔，FORTHというコンピュータ言語があった．これは，すべての演算をスタックで実行するアーキテクチャであり，ハードウェア化が簡単であったため，マイクロプロセッサにその処理系が実装されることが多かった．picoJavaも，FORTHと同じくスタックアーキテクチャを採用することで，ハードウェアによる実装を容易にしているものと思われる．

picoJavaは64個の32ビットレジスタをスタックとして内蔵する．Javaの各メソッドが実行されるたび

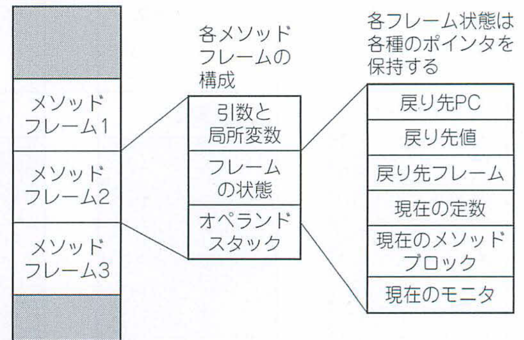


図1　picoJavaのスタックアーキテクチャ

メソッドAは多く
のスタックを使用
している
メソッドBを処理
する必要が発生
古いスタック内容が
ドリブラと呼ばれる
メモリに退避される
メソッドBが終了
すると内容が回復
される
スタック
にロード
不足している
スタック領域
メソッドA
メソッドB

図2　ドリブラシステム

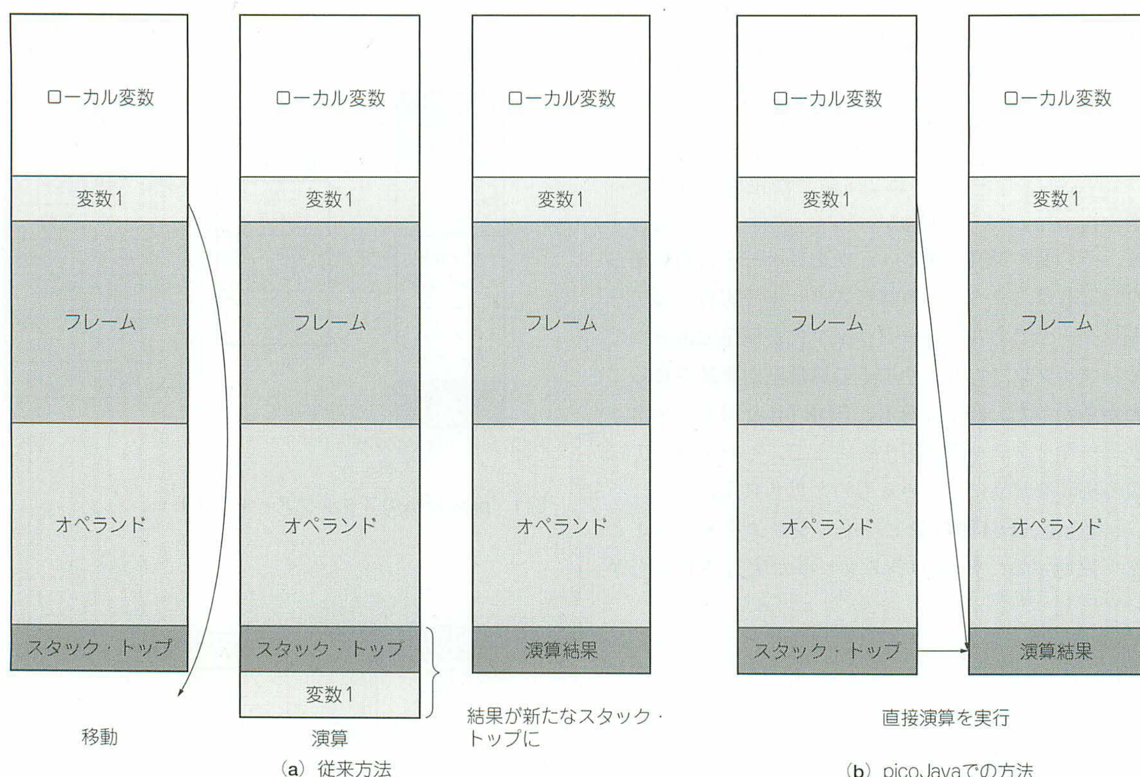


図3　スタック演算の高速化

に，64個のスタックからいくつかのスタック領域が確保されては使用される．もちろん，同時実行するメソッドの数が増えればスタックが一杯になってしまう．その場合は，ドリブラと呼ばれる専用メモリに古いメソッドのスタック内容をコピーし，空いたところを新しいメソッドに割り当てる．この入れ替え処理は，ドリブラシステムと呼ばれ，動的かつ自動的に処理される(**図2**)．

また，一般のスタックアーキテクチャでは，スタックトップにデータを移動し，スタックトップとその次とで演算を行い，スタックトップに結果を返す．

SunによるJavaコードの解析で，すべての演算の43%がスタック操作であり，スタックトップへのデータ転送が占める割合が多いと判明していた．このため，データをいちいちスタックトップに転送せず，スタックトップに計算結果を直接格納する方式を採用した(**図3**)．これにより，スタック操作の割合が29%に減少するという．このような命令の並列実行をフォールディングと呼ぶ．

なおpicoJavaコアでは，バイトコードを，フェッチ，デコード，実行，ライトバックからなる4ステージのパイプラインで処理する．またスタック(キャッシュ)へのアクセスは実行ステージで行われる．デコードステージでは，スタック操作の並列性を検出し，最大二つのスタック操作を同時実行することで性能を向上させる．picoJava-IIでは，この命令フォールディングが4命令に拡張された．

さらに，picoJavaではスタックにタグビットが付随しており，効率的なガベージコレクションを行うことができるらしい．詳細は不明だが，一般的には，命令のフォールディングと優秀なガベージコレクションがJavaを高速に実行するうえでのキーポイントである．

● picoJava-IIコアおよびMB86799の特徴

picoJava-IIコアの一般的なブロック図を**図4**に示す．特徴は次の通り．

- 32ビットプロセッサ：1MIPS/MHzの性能
- Javaのバイトコードを直接実行
- 従来のC/C++コードをサポート
- 6ステージパイプライン
- 最大4命令の命令フォールディング
- 命令キャッシュ：0～16Kバイト
- データキャッシュ：0～16Kバイト
- FPU(オプション)

これらにより，picoJava-IIコアは汎用のCPUより

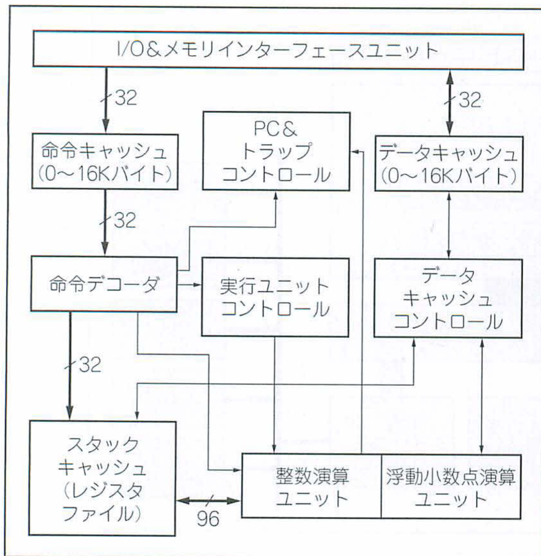


図4　一般的なpicoJava-Ⅱコアのブロック図

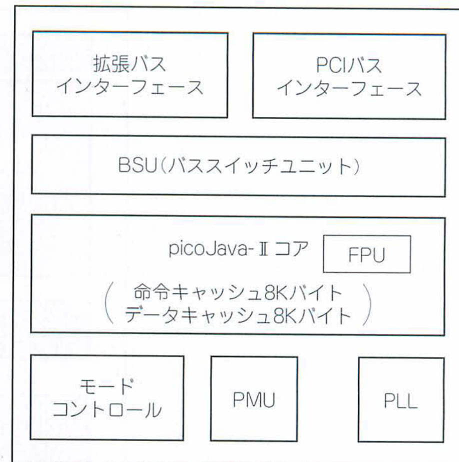
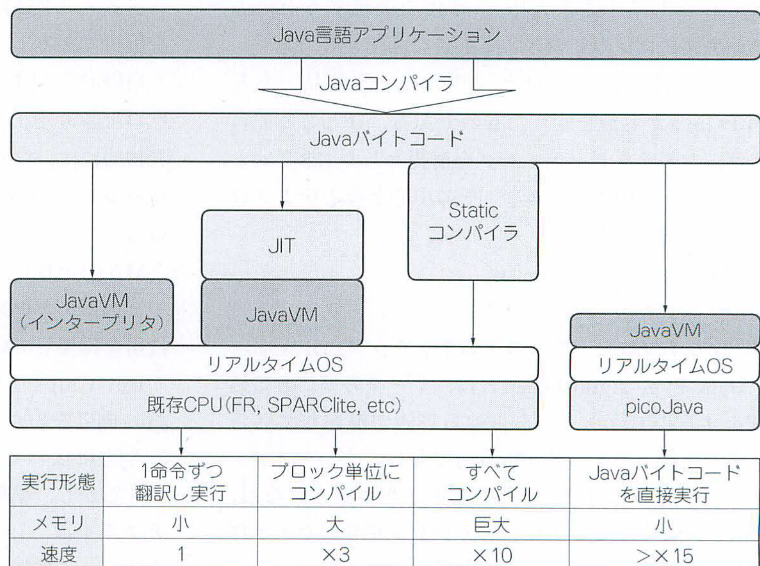


図6　MB86799のブロック図

も，Caffeine MarkというベンチマークにおいてMHz当たり約15倍のJava性能を発揮するといわれている．**図5**にJavaの各種の実行形態を示す．その中でもJavaプロセッサは最大の性能を発揮できる．

MB86799は，picoJava-Ⅱコアを採用した富士通のJavaプロセッサである．**図6**にMB86799のブロック図を示す．MB86799は，命令キャッシュ8Kバイト，データバス8KバイトのpicoJava-Ⅱコアを採用し，PCI 2.1に準拠したPCIインターフェース，SDRAM/SRAM/Flash ROM/ROM用の外部バスインターフェース，電力制御ユニットを内蔵する．0.25 μm CMOSプロセスで製造され，66MHz動作時に360mW@2.5V，40MHzで90mW@1.7Vという低消費電力を実現する．パッケージは256ピンQFPである．

富士通は，MB86799の拡販のために，J-REALOS/PJというITRONベースのソフトウェアパッケージと，J-StarterKitという評価ボードを用意している．確認するため富士通のWebサイトをチェックしたところ，「本デバイス(MB86799)は評価用です．量産はいたしませんのでご注意ください」とある．その代わり，周辺機能を簡略化したMB92901というチップが併せて紹介されていた(**図7**)．同WebではJavaプロセッサのロードマップも紹介されており，それによると富士通のJavaプロセッサは，2002年には本格的な



実行形態	1命令ずつ翻訳し実行	ブロック単位にコンパイル	すべてコンパイル	Javaバイトコードを直接実行
メモリ	小	大	巨大	小
速度	1	×3	×10	>×15

図5　Javaの実行方式のいろいろ

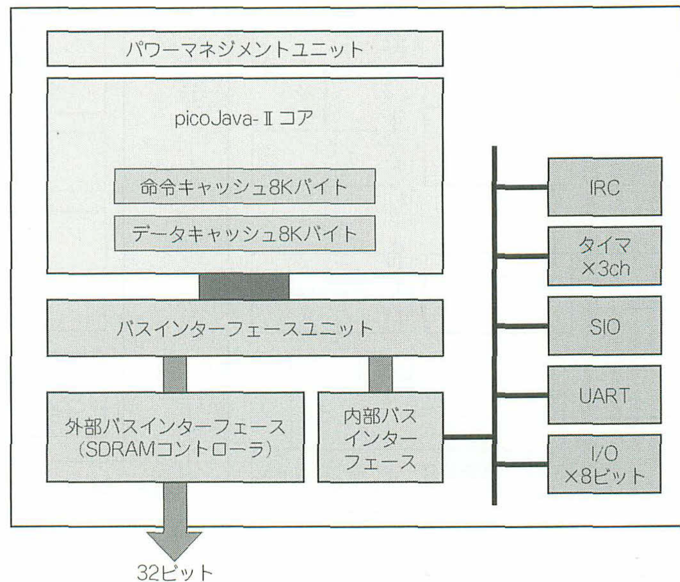


図7　MB92901のブロック図

実用化の予定だったらしいが….

● MAJC

1999年8月17日，SunはHot Chips XIシンポジウムにおいて，統合型マイクロプロセッサの基盤となる汎用アーキテクチャであるMAJC(Microprocessor Architecture for Java Computing：マジックと発音する)の技術詳細を明らかにした．MAJCは，マルチメディア処理ニーズの増大，Javaの普及，ネットワーク帯域幅の拡大という三つの業界トレンドを受けて登場した．名前からするとJavaを動作させるための専用プロセッサ技術ということだが，Sunにとってはそれ以上に「ポストPC」時代を実現するためのキーテクノロジの一つとして重要な意味をもつ．

MAJCは，Javaプロセッサの最上位製品であるUltraJavaが名前を変えたものである．ひとことで言えば，マルチスレッディングを前提とした128ビット長の4ウェイVLIWである．その中心となるテクノロジは次の三つである．

(1) Data Type Agnosticism
(2) マルチスレッディング
(3) マルチプロセッサシステムオンチップ(MPSOC)

Data Type Agnosticismとは，データの型ごとに処理をするのではなく，あらゆる形式が混在しているデータを扱えるようにする技術である．マルチメディアにおいては必須の技術になると予想される．なお，Sunは，音声やビデオなどディジタル化されたアナログ信号をナチュラルデータ型と呼んでいる．

マルチスレッディングとは，プロセッサ内部で複数のスレッドを並列実行する技術である．MAJCが採用するのは垂直型マルチスレッディング(Vertical Multi Threading)であり，これはあるスレッドがキャッシュミスなどで長時間待たされることが明らかな場合に，別のスレッドに実行制御を渡す方法である．これは，常に複数のスレッドを混合して実行する同時型マルチスレッディング(Simultaneous Multi Threading)と対極をなす．ハードウェア構造は垂直型のほうが簡単であるが，命令の実行効率は同時型が勝っている．Pentium4で採用されているハイパースレッディングは同時型マルチスレッディングのインテル用語である．

MPSOC(Multiprocessor-On-a-Chip)とは，複数のプロセッサを1チップに搭載するための技術である．具体的には，マルチプロセッサにおけるキャッシュコヒーレンシの維持を高速に実行する仕組みを有している．

MAJCは他にも，ギガビットクラスの高いI/O性能やディジタル信号処理機能をはじめ，数々の画期的な技術を採用している．

Hot Chips XIシンポジウムにおいては，「どこがJava向けなのか？」という質問が出たらしい．実際，MAJCにはバイトコードの直接実行という機能はない．しかし，単純でデータ形式を問わないこのアーキテクチャは，Javaのもつプロセッサ非依存性とマルチスレッド機能を完全にサポートすることができる．

表2　MAJC　5200の仕様

●2個の独立CPU
●動作周波数：500MHz
●動作電圧：1.8V(コア)，3.3V(I/O)
●共有，2ポートのデータキャッシュ16Kバイト
●800MHz，16ビット，Direct Rambus DRAMインターフェース
●66MHz，32ビット，PCIインターフェース
●64ビット，250MHz，NUPA(North Unified Port Architecture)
●64ビット，250MHz，SUPA(South Unified Port Architecture)
●テクノロジ：0.225 μm CMOS
●画像プリプロセッサ：リアルタイムジオメトリ圧縮，データパス
●パッケージ：624端子CGA(Column Grid Array)
●オンチップのデータ転送エンジン(中央クロスバースイッチ)
●バンド幅：2.0Gバイト/sec(NUPA，SUPAとも)，1.6Gバイト/sec(DRDRAM)
●性能(500MHz時)：6.16 GFLOPS，12.33 GOPS
●画像アクセラレータ
●オーディオプロセッサ
●メディアストリーム(オーディオ，ビデオ，スピーチ)のリアルタイム処理

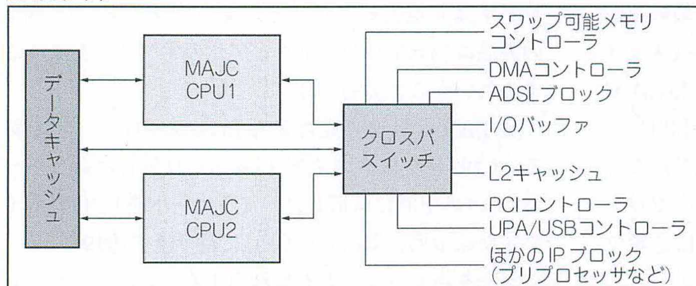


図8　MAJCのブロック図

そして，1999年10月5日には，MAJCの最初の実装である「MAJC 5200」の詳細が発表された．2個のCPU，メモリコントローラ，1チャネルの汎用I/Oコントローラ，2チャネルの高速I/Oコントローラ，画像プリプロセッサ，データ転送エンジン(DTE)，集中クロスバーを1チップに内蔵する(**表2，図8**)．

2個のCPUは500MHzで動作し，6.16GFLOPS，12.33GOPSという性能を実現する．これは，少し前のスーパーコンピュータを超える性能であり，ストリーミングや3Dグラフィックスなどの，いわゆるマルチメディアデータの操作に関する性能値も格段に向上すると予測される．ただ，消費電力は400MHz動作時に18Wと，それほど小さいわけではない．

ところで，MAJCはDSP機能をサポートする画像コプロセッサとして設計されている．単体の汎用プロセッサとしては利用できない．ソフトウェアのサポートも，現状は，SPARCのOSであるSolarisのグラフィックスライブラリとしてのみ提供される．ここらへんに理想と現実のギャップがある．結局，SunはJavaが高速に動くチップを諦めて，Java‘も’高速に動くチップに方向転換したのだ．

2 Javaアクセラレータ

既存のプロセッサに，Javaアプリケーションを高速に実行させるための機能を実装するという手法も考え出されている．一般的にこれをJavaアクセラレータと呼ぶ．次は各社から発表されているJavaアクセラレータについて説明しよう．

● Nazomi Communications社のJSTAR/JA108/JSMART

最近，Nazomi Communications社(以下，Nazomi)のJavaアクセラレータが話題になっている．Nazomi社は，2002年1月16日と3月13日にJavaアクセラレータの基本特許(US #6,338,160，US#6,332,215)を獲得し，積極的に自社製品のプロモーションを行っている．その内容は，RISCやCISCを利用したJavaプラットホームにおけるバイトコードの動的な最適変換を有する命令径路コプロセッサ(instruction path coprocessor：IPC)に関するものである．それを実現する製品には，JSTAR，JA108，JSMARTがある．

NazomiはJavaVMのハードウェアサポートに肯定

表3
JavaVMのハードウェア化の特徴

	MPUの拡張 (JSTARなど)	Javaアクセラレータ (JA108など)	Java専用 プロセッサ
システム透過性	あり (命令無変更が条件)	あり	なし
デバイス/システムの組み込みに要する労力	低	低	高
JVMを適用する労力	低	低	高 専用JVMが必要
デバイス/システムの市場への提供速度	種々	速い	遅い 専用システム設計
Javaソフトウェアの性能	良い	最高に近い	最高に近い
消費電力	最高	かなり良い	良い

的である．某調査会社の「2005年までにはJavaVMの70％以上が何らかのハードウェアの助けで高速化される」という報告が拠り処である．そのための条件として，ソフトウェアによる高速化のコスト(メモリ増設などによる)よりも低価格であること，JavaVMを初めとするソフトウェア環境を変更なしで利用できること(これをトランスペアレント＝透過性と呼んでいる)が必要と考える．現在，JavaVMのハードウェア化による高速化に関しては，次の3種類に分類できる．

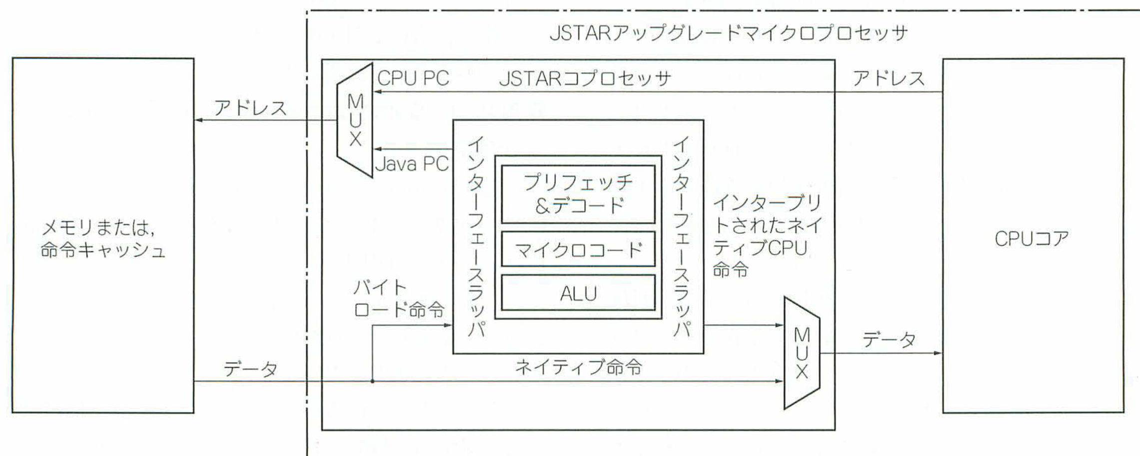
(1) バイトコードインタプリタの内蔵と命令セットの拡張
(2) 独立チップ，あるいは，SoC部品としてのJavaアクセラレータ
(3) ネイティブ命令としてJavaのバイトコードを用いる専用Javaプロセッサ

それぞれの方式の特徴を**表3**にまとめる．Nazomiは，(1)にJSTARとJSMARTを，(2)にはKchipファミリ(最初の製品はJA108)を提供する．(3)のアプローチは非現実的として考えない．つまり，Nazomiの製品はJavaVMハードウェア化のすべての要求に応えるということであろう．

JSTARは論理合成可能なJavaコプロセッサである．CPUとシステムメモリ(あるいは外付け命令キャッシュ)の中間に位置し，バイトコードをCPUのネイティブ命令列に変換してCPUに受け渡す(**図9**)．

データキャッシュを有効利用するために，キャッシュ経路は**図10**のように構成する．JSTARは228種あるバイトコードのうち159命令を直接実行する．それ以外はホストCPUによるソフトウェアエミュレーション(従来のJavaVM)に任せる(**図11**)．

また，**図12**に示すように，JSTARアーキテクチャはJavaバイトコードをデコードして最適化されたネイティブCPUの命令列を生成するのに2段パイプラ



JSTARは，メモリからJavaバイトコードを読み出し，それをネイティブCPU命令の最適シーケンスに変換する際，メモリへのアクセスを最小限に減らし，システム全体の消費電力を低減する

図9　JSTARの構成図

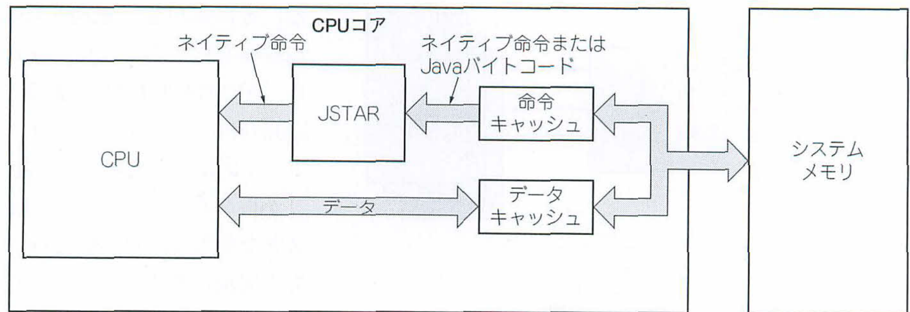


図10
JSTARTのキャッシュ経路

インを用いる．Javaの中間言語は可変長なので，単純には命令変換ができないためである．

まず，プリフェッチユニットが命令ストリームの整列とバッファリングを行う．いったん命令がバッファリングされると，IDP(Instruction Decode, and Parallelism)ユニットがそれらの命令に対しコンパイラと同様なフォールディング規則を適用して，最適化された命令列を生成する．その後，特許取得のSTB(Stack and Translation Buffer)ユニットが，スタックトップや局所変数のキャッシュといったJava専用スタックの管理を行うとともに，命令列をCPUに引き渡す．

JSTARを使用すると，従来のソフトウェアによるJavaVMと比べて10～40倍の性能が得られる．消費電力も従来の95％に削減できるという．構造的にはシステムメモリとCPUの間にロジックを挿入することになるので，外部には追加端子は必要ない．つまり，従来品とそのまま差し替え(ドロップインリプレース)が可能なのである．JSTARの存在はソフトウェアからは見えないので，既存のソフトウェアや開発ツールがそのまま流用できる．

理論的にはすべてのJavaVMに適用できるが，現在サポートしているのはPersonalJava，CVM，KVM(J2ME-CDC，J2ME-CLDC)である．適用可能なホストCPUとして，ARMではARM7，ARM9シリーズ，

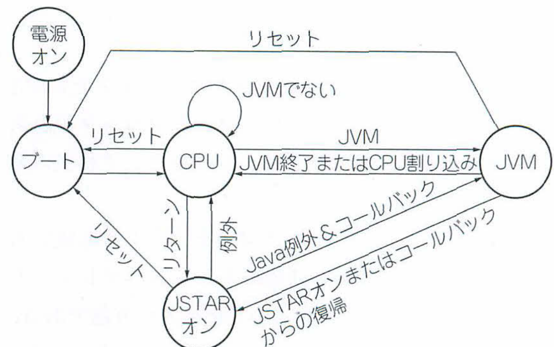


図11 JSTARTの状態遷移

MIPSでは4Kシリーズ，Lexra(MIPSアーキテクチャ)ではLX4180/4189/4280を想定している．

JA108は，NazomiのKchipファミリ最初の製品である．JSTARがVerilog RTLで供給されるIPコアであるのに対し，こちらは単体のアクセラレータチップである．10×10mmで128ピンBGA，7×7mmで128ピンBGAで提供される．機能的にはJSTARとほとんど同じであり，ホストCPUやSoCとSRAMに似た双方向インターフェースで実装する(**図13**)．このため，簡単にシステムのメモリバスに追加できる仕様になっている．

小型ハンドヘルド機器，特に無線機器のようなJ2ME-CLDC(KVM)を利用する小型機器を対象にす

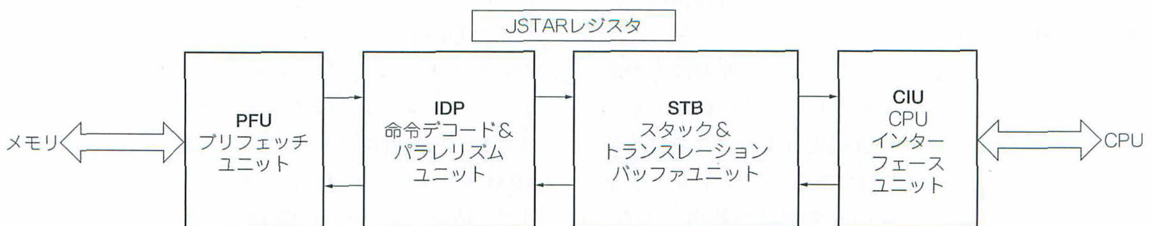


図12 JSTARTのパイプライン

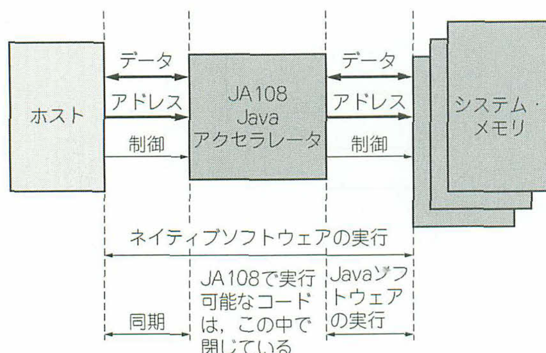


図13　JA108の構成図

る．JA108は，従来のJavaアプリケーションを200倍に高速化する．このために，システムクロックの高速化，メモリ増設，新規ツール，ソフトウェア移植の努力は不要である．

JSMARTは，JSTARのスマートカード対応版である．JSTARやJA108と同様のトランスペアレント方式を採用し，ドロップインリプレースが可能である．違いは，JavaCard2.1.1仕様に対応していることと，RAM/ROM/フラッシュのメモリ管理機構を内蔵していることである．JSMARTを使用することで，Javaカードの実行速度を10～40倍に高速化する．また消費電力を95％に削減する．

2003年9月17日，ルネサステクノロジ(旧：日立製作所と三菱電機)とNazomiはJavaプログラムの高速実行技術で提携した．ルネサステクノロジは，Nazomi社が提供するJavaアクセラレータを自社のCPUコア「SH-Mobile」に内蔵する．これは携帯電話でJava性能の高速化が必要と判断されたためと思われる．ルネサステクノロジによれば，SH-Mobileに搭載するJavaアクセラレータとして複数の技術を検討したところNazomiの技術がもっとも優れていたという．

ちなみに，Nazomiとは新幹線「のぞみ」が由来になっているらしい．Nazomiの前身であるJEDI Technologiesは，明らかに，STAR WARSのジェダイの騎士が由来であろう．ベンチャ企業の名称は面白い．

● ARM社のJazelle

おそらくJavaは生き残るが，各社の努力にもかかわらずJavaプロセッサは絶滅すると思われる．その場合でも，バイトコードを直接実行することで得られる10倍以上の性能は魅力的である．バイトコードを直接実行する汎用CPUという発想は当然出てくる．先に述べたIBMのVLIWプロセッサも同じ発想である．その実用化に成功したのがARMプロセッサのJazelle拡張である．JazelleはARM9で使用可能である．ARM10以降は標準装備になると思われる．

Jazelleの方式は，バイトコードの直接実行，ソフトウェアによる実現，未実装例外発生の三つのカテゴリからなる．バイトコードは，4個のスタック用レジスタを使用し，140種を直接実行する．ソフトウェアでの実現に使用するための特別なJava専用命令は存在しないようである．

ARM社によるARM9EJでのCaffeine Markのシミュレーションでは，

- 通常のJavaVM　　　　… 0.7CM/MHz
- ソフトウェアの最適化 … 1.7CM/MHz
- 専用Javaプロセッサ　 … 2.9CM/MHz
- Jazelle(ソフトウェア最適化を含む)
  … 6.0CM/MHz

という結果を得ている．実に，Javaプロセッサの2倍以上の性能を得ることができる．また，Jazelleは消費電力の面でも有利で，Jazelleを使用しない場合の1/7で済むという．

JazelleはJavaのバイトコードを直接実行する技術である．全体の95％のバイトコードがハードウェアで実行可能である．Javaの分岐を静的および動的な分岐予測で高速化する．バイトコードの抽出とデコード処理がパイプラインの中に取り入れられ，ネイティブコードと同等に実行される．その方式は，命令キャッシュとデーコーダの間でネイティブコードへの変換を行うJSTARの方式と酷似している．しかし，ARMはそれとは別のオリジナル技術だと主張している．

しかしというか案の定，2002年5月28日にNazomiはARMをJava特許(US#6,332,215)を侵害しているとして北カリフォルニアの連邦地方裁判所に提訴した．Jazelleを潰してしまえば，自社技術(JSTAR)をARMが採用せざるを得なくなるという思惑があるのだろう．今後の推移が注目される．

2002年になって，ARMのJazelleの説明が微妙に変更されている．Javaのバイトコードは，ARMネイティブ，Thumbに続く第3の命令セットであるとし，命令デコードステージで解釈されるとしている．つまり，パイプラインのデコードステージの前にバイトコードからの変換ステージが挿入されるのではなく，ARMネイティブやThumbと同じレベルでデコードされ，直接実行される(**図14**)．これは件の特許対策でマイクロアーキテクチャの変更が行われたものと考

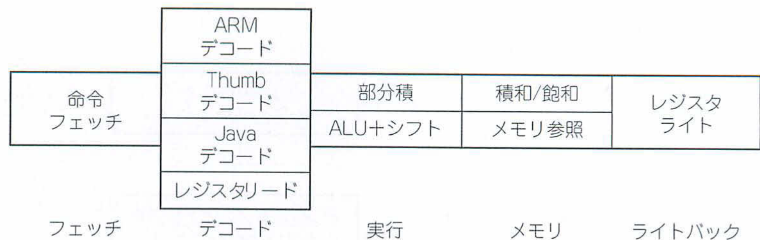


図14 ARM9EJ-Sのパイプライン

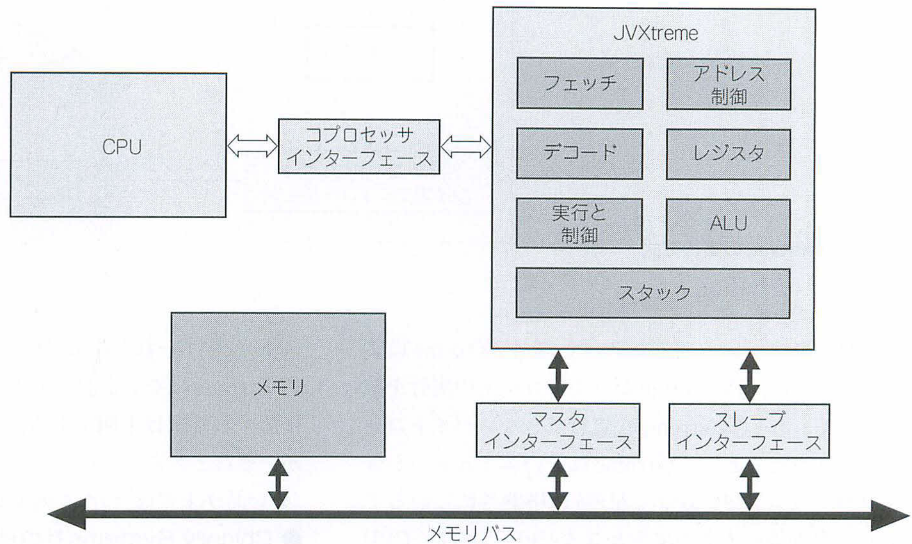


図15
JVXtremeのブロック図

えられる．

それにしても，Nazomiにしろ，後述するInSiliconにしろ，ARMと共に実行することを目指しているのが興味深い．やはり，ARMの「組み込みIPコアのシェアが最大」というのが魅力なのだろう．

● InSillicon社のJVXtreme

JVXtremeはInSillicon社のJavaプロセッサである．その前身であるJVXのほうが有名であるが，基本構造は同じであると推測される．JVXtremeはJazelleやJSTARとは異なるアプローチを取る．JazelleやJSTARがバイトコードをデコードして，ネイティブコードに変換し，ネイティブコードとして実行するのに対して，JVXtremeはバイトコードを直接実行する（**図15**）．ユーザーにはその違いは分からないが，JVXtremeではサポートしないバイトコードを実行してくれるCPUが必要である．その意味でのコプロセッサ（Javaエンジン）である．リリース形態としては，SoC用のIPコアとCPUのメモリ空間にマップされて動作する単体チップ形式がある．

JVXtremeを一言でいうとスタックベースのJavaエンジンである．結局，picoJavaと変わりない．しかし，picoJavaとは異なり，直接実行できるバイトコードが92種とかなり少ない．これは，統計的に実行頻度の多い命令のみをハードウェアでサポートすることにしたためであり，性能的にはこれで十分と言われている．

また，ハードウェアスタックの深さを論理合成時に指定できるのが特徴でもある．スタックのオーバフローやアンダフローは自動的に処理される（picoJavaのドリブラのようなものか）らしいが，スタックオーバフローは実行速度に影響を与えるので最適な値に構成するのが望ましい．その目安として，ハノイの塔の実行時にスタックオーバフローの確率は24段で1%未満，32段もあればほとんどすべてのプログラムでオーバフローは発生しないとされている．これは一つのメソッドしか存在しない場合の話であろう．実際のJavaプログラムでは，複数のメソッドが並列に動くので，もう少し深いほうが有利だと思う．事実，picoJavaでは64段だった．

JSTARやJazelleのアプローチとは異なり，JVXtremeを使用するためにはJavaVMを変更する必要がある．**図16**に示すように，CPUは，まずJavaのバイトコー

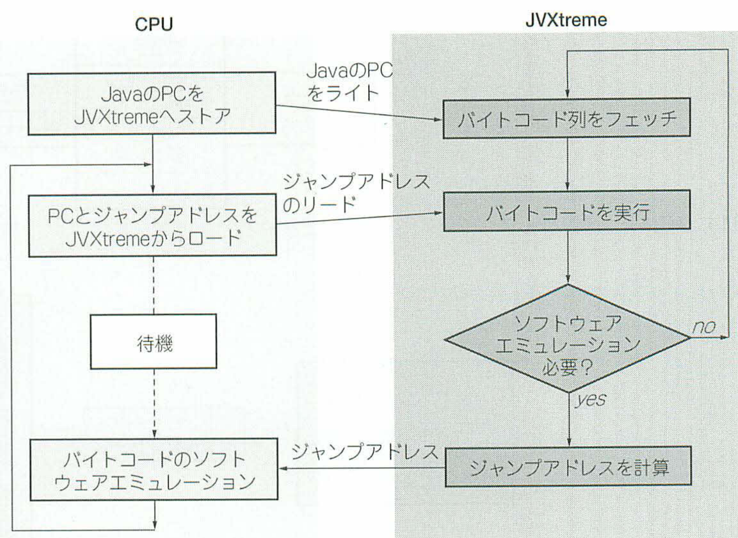


図16　JVXtremeの実行方式

ドが格納されているアドレス(PC)をJVXtremeに渡す．その後は，JVXtremeがバイトコードの実行を続けていく．もしJVXtremeが実行できないバイトコードに行き当たると，JVXtremeはそのエミュレーションプログラム(CPU命令で記述)が格納されているアドレスを計算し，そのアドレスをCPUに戻す．CPUはそこにジャンプすることでバイトコードをエミュレートする．エミュレーション終了後はJVXtremeが新しいアドレスを与えてくるまで待ちに入る．その間，JVXtremeは並行にバイトコードを実行している．

現在，InSilicon社はARM7とARM9用のIPコアを用意している．また，ARMのAHB(Advanced High speed Bus)に直結するメモリマップ型のコプロセッサも存在する．この類のアクセラレータではCPUとの通信速度が性能に影響するが，ARMの場合，JVXtremeは，CPUからバイトコードのアドレスを与えられてから実行を開始するまで5クロックかかるという．

JVXtremeの動作速度は，0.18 μmプロセスで製造された場合は200MHzというから，まずまずの性能であろう．InSilicon社は15～55倍の性能が得られるとしている(JVXでは10倍の性能だった)．Caffein Markで約1300CMというから，Jazelleと同程度の性能である．

JVXtremeは，JazelleやJSTARに対する新機軸としてマスコミに取り上げられたが，結局はpicoJavaのアーキテクチャに戻っただけである．

InSiliconのWebサイトには利用可能なIPコアのリストが掲げられているが，その中にJVXはあるがJVXtremeはない．JVXに関してもIPコアリリースに関する詳細は不明である．利用にNDAが必要なのか，それとも未だにロジックがフィックスしていない(早い話が未完成)のか不明であるが.

● Chicory Systems社のHotShot

picoJavaを嚆矢とし，Jazelle，JSTAR，JVXtremeに続く，Javaアクセラレータの第3の新機軸として登場したのが，Chicory Systems社のHotShotである．他社の方式がバイトコードの直接的な高速実行であるのに対し，HotShotは別の手法を採用する．それはハードウェアによるJIT(Just In Time：動的コンパイラ)の実現である．つまり，バイトコードを動的にホストCPUのネイティブコードに変換し，その実行はホストCPUに行わせる方式である．

この手法は，Transmeta社のCrusoe(x86命令を専用VLIWに変換)やTransitive社のDynamite(x86やARM命令をPowerPCやMIPS命令に変換)と同じである．性能的には，ソフトウェアによるJITと同程度と思われるが，Chicory Systems社はバイトコードのソフトウェアインタプリタを使用する場合の25倍の性能が得られるとしている．

HotShotもホストCPU(ARMを念頭に置いているらしい)のコプロセッサとして動作する．その最大の特徴は，バイトコードをネイティブコードにコンパイルするために種々の最適化を行うことができることである．他のバイトコードのアクセラレータで最適化といえば，命令フォールディング程度しかない．現実に

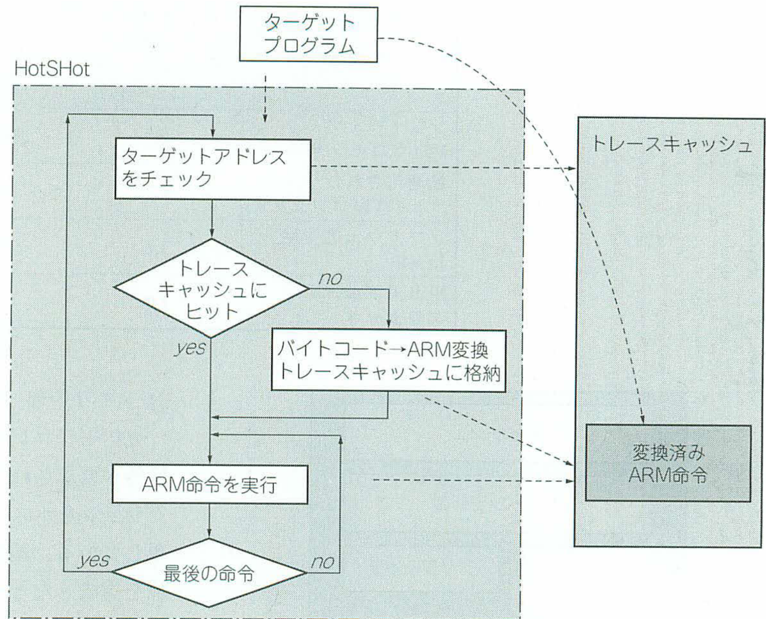


図17 HotShotの動作概念

は最適化ハードウェアを備え，送られてくる命令列を常に解析し続ける．最適化の手法は，ヒューリスティック（評価関数＝早い話が経験論）に基づいた高度なレジスタ割り付け，関連する演算を一つに結合して1命令として実行する特殊な技術，分岐の最適化である．

HotShotは，Javaのメソッドやメソッドの一部分を単位として，1クロックに一つのネイティブ命令を生成できる．また，バイトコードの冗長な実行をなくすために，主記憶上にLRU方式のトレースキャッシュ（2K〜128Kバイト）を設ける．これは，Pentium4の命令トレースキャッシュやCrusoeのトレースキャッシュのようなものと考えればいい．つまり，同じような処理を何度もコンパイルすることはせず，以前にコンパイルした結果を使用することで，コンパイル速度を稼ぐのだ．バイトコードの実行形式は**図17**のようなものと想像される．

HotShotの存在意義に関しては議論のあるところである．わざわざライセンス料を払って，既存のソフトウェアで書かれたJITを専用プロセッサで置き換える価値があるか否かである．ただ，現時点でのChicory Systems社の状況は不明である．そのWebサイトのURL（`http://www.chicorysystems.com/`）は既に存在していない．

● Zucotto Wireless社のXpresso

世界的な傾向として，携帯電話に複雑なアプリケーションを実行させることが流行りである．そこで，携帯電話メーカーは自社製品でJavaのサポートを表明している．しかし，現在のJavaVMの速度は，通信機能の余剰能力で動作していることもあり，動作が非常に遅い．そこで，組み込み制御分野での専用Javaプロセッサというソリューションが生まれてくる．

Zucotto Wireless社は，ソフトウェアの最適化や専用ハードウェアによるアクセラレータよりも，Javaプロセッサは，

- 単位電力当たりの性能
- 周辺機能として1チップに集積することの容易さ

の点で利点があるとしている．

そこで，Zucotto Wireless社はXpressoというJavaプロセッサを開発した．これは，Javaプログラムの性能ボトルネックを調査し，性能に効く部分に着目して，チューンナップを行っている（**表4**）．

図18にXpressoのブロック図を示す．拡張や種々のシステムへの対応を容易にするために，マイクロコード制御のCISCプロセッサになっている．Xpressoの特徴は次のとおり．

- 32ビットCISCプロセッサ
- バイトコードを直接実行
- 最適化されたスタックおよび局所変数キャッシュ
- 単一命令/データキャッシュ
- RAMに格納されたマイクロコード制御
- 5ステージパイプライン
- 静的な分岐予測

表4
Java性能のボトルネック

実装方式	インタプリタループのオーバーヘッド（クロック数）	Invoke（クロック数）	イベント/ネイティブ（クロック数）	メモリ管理
ソフトウェアVMプロセッサ	20	300以上	33	停止してメモリ走査
最適化されたソフトウェアVMプロセッサ	6.7	300	10	停止してメモリ走査
ソフトウェア VM Jazelle	1.1	300	33	停止してメモリ走査
SLICE Xpressoプロセッサ	1.0	35	1.0	スムーズ

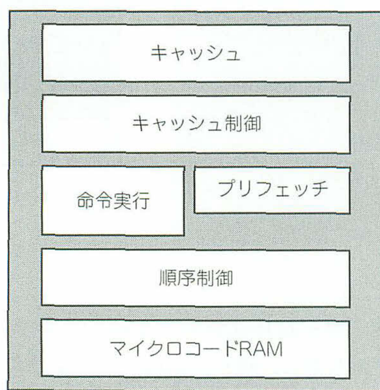


図18　Xpressoのブロック図

- ゲート規模：60Kゲート
- ほとんどの命令を1クロックで実行
- 拡張性の容易さ

表4に示すボトルネックの解消はマイクロコードにより実現しやすくなっている（Xpressoは，OS，ドライバ，割り込み処理ルーチンを一手に引き受けるSLICEというソフトウェアと共同で実行されている）．つまり，高速処理のため，マイクロコードで次のような機能をサポートする．

1) Invoke処理

Javaにおいて，Invoke（メソッド呼び出し）は頻繁に出現し非常に複雑であり，効率的なアプリケーション実行のためにはこの処理の高速化は特に重要である．

- invokevirtual
- invokestatic
- invokeinterface

2) 外付け周辺，イベント処理，ネイティブ資源へのインターフェース

Javaプログラムから，外部I/Oなどネイティブな資源へのアクセス時間の短縮や，直接的なイベント処理（物理インターフェース）．直接ハードウェアにアクセスできる機能を実現する．

3) メモリ管理

ガベージコレクション性能とメモリ管理ルーチンの性能が要求される．インクリメンタルなガベージコレクションをホストプロセッサの機能を邪魔しないで実現している．結果として，Xpressoは次のような高性能を達成したという．

9　CaffeineMarks/MHz

22.5　CaffeinMarks/mW

これは，性能的にはARM9のJazelleの約2倍，JSTARの約3倍の速度である．省電力的には，ARM9のJazellの約4倍，JSTARの約8倍の効率である．

● parthus社のMachStream

parthus社は，ハンドヘルド機器やモバイル機器のインターネット化に着目する．いきおい，それらの機器でも複雑なアプリケーションを実行する必要性が生じる．parthus社のMachStreamはそのような要求に応えるコプロセッサである．**図19**は典型的なモバイル機器向けASICのブロック図である．MachStreamはシステムバスに結合し，一つの周辺機能として動作する．従来との互換性を保ちつつ，集積を容易に行えるのが売りの一つである．

MachStreamとは，Javaアクセラレータだけではなく，メディア処理などの複数のアクセラレータと並列に接続できる．**図20**はJavaアクセラレータを装備した場合のMachStreamのブロック図である．その動作は次のようになる．その実現方法は，Chicory Systems社のHotShotに酷似している．

(1) 特定のJavaコードブロックを高速実行するようにJVMソフトウェアから要求される．
(2) Javaアクセラレータは変換キャッシュディレクトリをチェックし，すでに生成されたコードを再利用できるか否かを調べる．
(3) 生成されたコードがキャッシュされていなければ，Javaアクセラレータは Javaコードをフェッチ，最

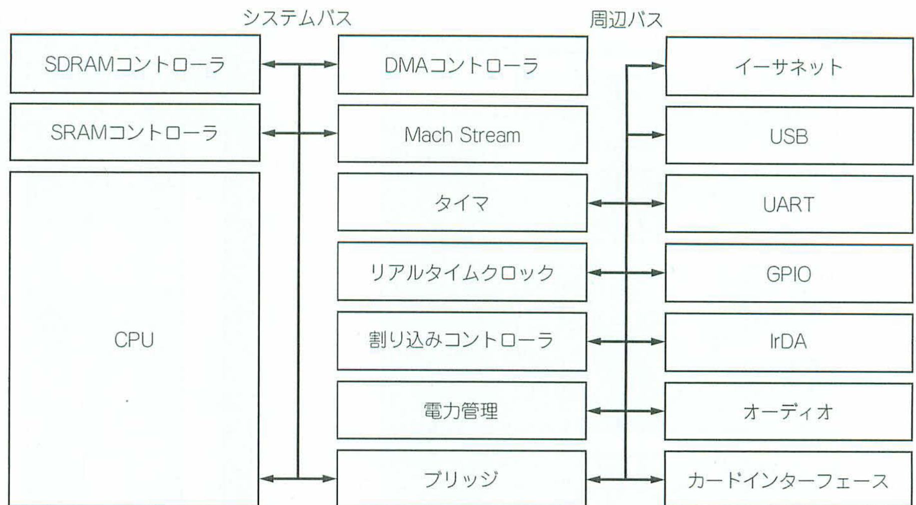


図19
典型的なモバイル機器向けASICとMachStream

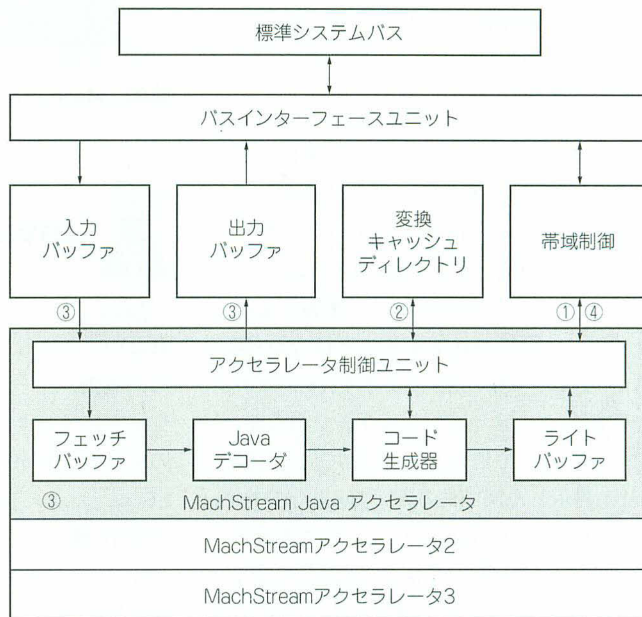


図20
MachStream Javaアクセラレータ

適化されたネイティブコードの塊を生成，その結果をメインメモリの変換キャッシュ領域にライトする．

(4) コードのポインタがCPUで動作するJVMソフトウェアに返され，そのコードが実行される．

図21にMachStreamの処理性能，**図22**に電力性能を示す．これらは，ARM920コアにMachStreamを集積した場合の性能であり，ソフトウェアのみの実行の35倍の性能を得ることができる．電力性能は約10倍以上である．

● NanoAmp TechnologyのMOCA-J

2003年2月12日，NanoAmp Technologyは，MOCA-Jという携帯電話向けJavaアクセラレータでJ2MEの実行速度をソフトウェアのみの処理の20倍に向上させたと発表した．処理性能が速くなっても電池寿命は据え置きのままというのが売りである．

ほかのJavaアクセラレータと同じく，メモリと同じようにアクセスできる．メモリ速度を稼ぐため，メモリチップとMOCA-Jを1チップにMCP(Multi-Chip Package)形態で封入する(**図23**)．最近流行りの言葉で言えばSIP(System In a Pakage)である．また，この形態をSMARTcombo-Jと呼ぶ．

MOCA-Jアクセラレータは，JavaVMで定義されている227のバイトコードのうち206種を直接(ほとんど

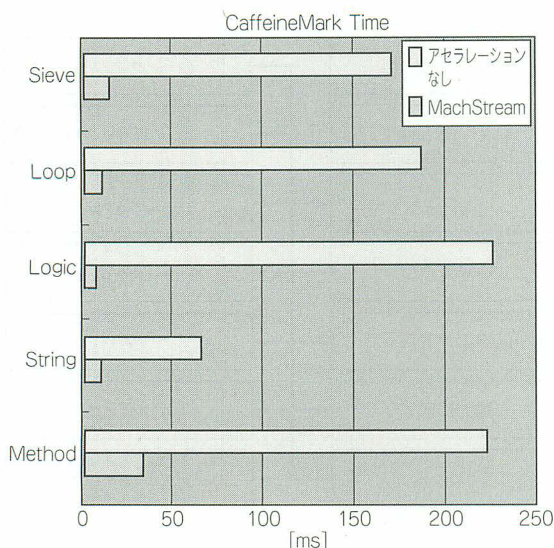


図21　MachStreamの処理性能

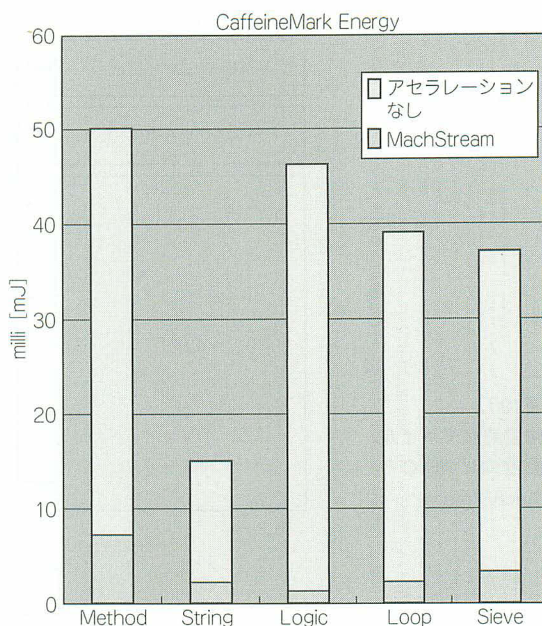


図22　MachStreamの電力性能

の場合)1クロックで実行する．

ARMのJazelleや他のJavaアクセラレータは，一般の命令コードとJavaのバイトコードを共通のシステムバスから取り込むため，Javaアクセラレータへのバイトコード供給量が低下する．SMARTcombo-JはJavaコードを取り込むための専用バスを備えるため，バイトコードの供給量が低下することはない(**図24**)．NanoAmp社は，「JazelleによるJavaアプリケーションの性能向上はせいぜい3～5倍であるが，SMARTcombo-Jならば20倍だ」と豪語する．**図24**はNanoAmp社が提示した図であるが，Jazelleがバイトコードの翻訳(On the fly interpretation)であると説明している．現在，ARM社はJazelleは翻訳ではないと強調しているが，ここではそのまま示しておく．

MOCA-Jは，現在ダイ形態(裸のシリコン)でサンプル配布中であり，2003年の第二四半期に量産予定である．MOCA-JをMCP形態にするのにかかる費用は5ドルを超えないという．実際の価格は要交渉ということらしい．

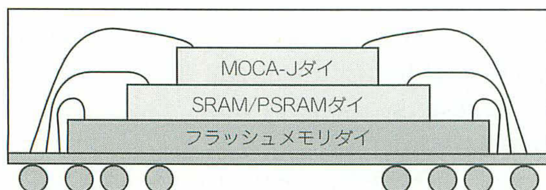


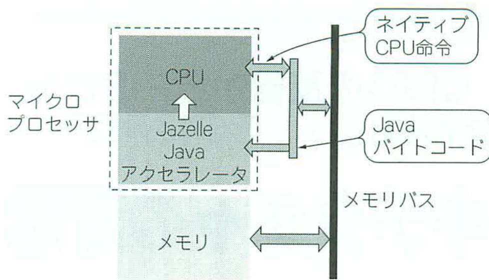
図23　SMARTcombo-Jの構造

3　Javaプロセッサの今後

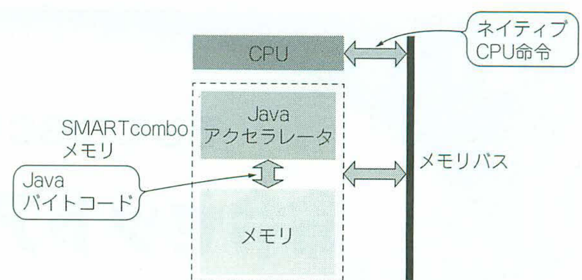
Javaプロセッサの存在を知らない人は多いかもしれない．しかし，誰もが発想するアイデアには違いない．実際，ハードウェアによるJavaの高速化をキーワードにインターネットで検索すると，かなりの種類のプロセッサが提案されているのがわかる．たとえば，http://www.jdance.com/shop/hardware.shtmには16種のJavaプロセッサが紹介されている．この中でよく言及されるのが，先に挙げたJazelle，JSTAR，VMXtremeである．しかし，根本的な実装方式(思想)はpicoJavaの拡張(サブセットかも)に過ぎない．バイトコードの実行がCPUのパイプラインに組み込まれているか，コプロセッサとして動作するかである(HotShotやMachStreamのような，動的なJIT方式もあるが)．

ところで，Javaプロセッサは本当に必要なのかというのは，Javaプロセッサの発表当時から頻繁に繰り返されてきた議論である．LISPやSmalltalkの経験から，専用チップを開発するよりも汎用のRISCでソフトウェアのチューニングに注力するほうがいい性能を出している．

また，一般のCPUベンダは既存の汎用CPUでもJavaを効率良よく処理できると主張している．たと



(a) Jazelle オンザフライインタプリタ



(b) SMARTcomboダイレクト実行

図24　SMARTcombo-Jは専用バスを備える

えば，ARMはStrongARMのアーキテクチャをJavaやPostScriptなどのようなスタックベースの言語の組み込みアプリケーションに最適化している．複数レジスタをスタックフレームに一括して退避/回復する命令がそれに当たるという(これはJava用命令というわけではないが)．

さらに，MIPSはメモリのバンド幅を上げることでJavaの実行を高速化できると主張している(最近では，浮動小数点演算の高速化が必須ともいっているが…)．経験的には，キャッシュの構成を工夫してヒット率を向上することができれば，一般のRISCでもJavaVMやマルチスレッドを高速に実行できる．Javaプロセッサの提唱者であるSun自身が，MAJCで一般のプロセッサを目指しているところにその凋落を予感する人もいるだろう．

Javaプロセッサの存在意義の拠り処は，Javaがインターネットと密接な関連があるため爆発的に普及するであろうという予測である．しかし，そういう文化的背景を掲げながらも，Javaプロセッサは当初から組み込み制御分野を目指している．インターネットでの応用では，専用プロセッサを作ったとしても，PCやEWSの進化によってすぐに性能的に追い越されてしまうことが最初からわかっているような主張である．

ところが，組み込み制御分野に注目してみた場合，Javaの言語仕様ではリアルタイム制御に向いていないとして，Javaの使用に懐疑的な開発者も多い．反面，Javaを用いれば，セキュリティの保護や個人認証を簡単に実現できるという主張もある．つまり，ICカードや無線LANでの応用が見込める．しかし，Javaに利点を見出す人々も，汎用CPUのほうが融通が利くと主張する．C/C++を高速に処理できるCPUは，当然，JavaVMも高速に処理できると．逆もまた真なりで，JavaVMを高速化するためにはC/C++を高速に処理できるアーキテクチャにすればよい．これは一般のCPUの方法論である．セキュリティ実現のためのトレンドは，Javaを高速化するよりも，もっと低レベルで基本的な機能である暗号のサポート機能などを汎用CPUの命令セットに実装して対応する傾向にある．

しかし，携帯電話での応用は未知数である．NTTドコモの503iで一般化すると思われたJavaプロセッサであるが，504iでは，Java以外の応用にも対応できるようにとJava専用ではないアプリケーションチップを搭載するのが流行のようだ．しかし，今後，さらなるJava性能が必要となる場面もあるかもしれない．

まとめ

結局，Javaプロセッサとは何だったのだろうか．Sunの一人相撲だったのか．Javaの実行をハードウェア化するという発想は多分間違っていなかった．コプロセッサとしてしか提供できなかった点が敗因と思われる．CPUと別チップというのであれば，コストパフォーマンスが悪過ぎる．

将来，CMP(Chip Multi-Processor)技術が進んでくればJavaプロセッサ(IPコアとして)の復権はあるかもしれない．つまり，CPUとJavaプロセッサを1チップに集積した形態である．これは，Nazomi社，Zucotto Wireless社，parthus社との思惑とも一致する．それまでは，Jazelleのように，CPUがバイトコードを直接実行する方式が流行るのだろう(あれ？V850+Jの立場は？)．いや，GHzを超えるクロックのCPUなら，何もしなくてもJava程度はサクサク動かせるという意見が大勢か．明日は明日の風が吹く？

第11章

CISCの反省からRISCへ，そしてRISCもまた複雑化し，その将来は……

命令セットアーキテクチャの変遷

コンピュータの誕生以来，さまざまなアーキテクチャのMPUが登場してきた．本章では，究極のCISCと呼ばれるV60/70やTRONチップ，そしてRISCの代表であるMIPS，ARM，i860，88000，SPARC，PowerPC，PA-RISC，Alphaの命令セットアーキテクチャについて解説する．豊富な命令数とアドレッシングモードを備え複雑化したCISCの反省からRISCが生まれ，そしてRISCもまた複雑化していくようすがわかるだろう．

CISC(Complex Instruction Set Computer＝複雑な命令セットのコンピュータ)という言葉は，命令セットが複雑な昔のコンピュータの命令セットを揶揄(やゆ)した，**RISC**(Reduced Instruction Set Computer＝縮小された命令セットのコンピュータ)の研究者が創造した言葉である．CISCとRISCの**命令セット**には，どのような違いがあるのだろう．本章では具体的なMPUの命令セットについて解説することで，それぞれの特徴をみていきたい．そして，MPUの進化や変遷につれて命令セットがどう変わっていったのかを知っておこう．それは，とりもなおさずMPU自体の歴史ともいえる．

1 コンピュータアーキテクチャとは

● IBM System/360の時代

コンピュータアーキテクチャとは何だろうか．実はコンピュータアーキテクチャという言葉が初めて使われたのは，それほど古くない．1964年，Gene M. Amdahl氏らがIBM Journalに寄稿した論文である"Architecture of the IBM System/360"の中なのである．

そこでの定義は，プログラマから見たコンピュータということ．つまり，命令セットと命令セットの実行モデルということだった．その本質は，アーキテクチャの設計と特定の実装方式を切り離して考えることにある．同じアーキテクチャをもつコンピュータは「ファミリ」と呼ばれ，同じファミリ内であれば，ハードウェアの実装方法やファームウェアが異なっても，プログラムに互換性がある．プログラマは命令セットだけを気にしていればよい．この概念はファミリという考え方を一般的にし，IBM System/360やSystem/370だけでなく，PDP-11やVAX，680x0，x86アーキテクチャの開発に大きな影響を与えた．

● コンピュータの方式を示す大きな概念

しかし技術の発展により，Amdahl氏らの定義は古くなってきた．プログラムの実行はライブラリ，OS，システム構成に影響され，命令セットが同じであっても互換性があるとは限らない．いまや互換性というのは，OSとのインターフェースやさまざまな規格を統一しないと実現できない．また，アドレス空間のビット数，仮想記憶やキャッシュの構成などの実装方式も互換性に影響を与えることがある．

この意味でアーキテクチャという言葉は，現在ではコンピュータの方式を示す非常に大きな概念になっている．そのため，特定の方式に言及する場合は，命令セットならば**命令セットアーキテクチャ**，実装方式ならば**マイクロアーキテクチャ**(ハードウェアアーキテクチャ)，システム構成なら**システムアーキテクチャ**などと，固有の名称を使用するようになっている．

さて，本書の第1章から第6章までは，おもにコンピュータのマイクロアーキテクチャについて解説してきたが，ここでは命令セットアーキテクチャに注目する．

● 基本的な命令機能はどれも同じ

MPUの命令セットアーキテクチャの基本はどれも

同じである．データの移動，データの加減乗除，論理演算(AND，OR，XOR，NOT)，比較と条件分岐命令である．これらは，整数データや浮動小数点データを処理する演算としてかならず定義されている．場合によってはNOTがNORであったり，比較と条件分岐が一つの命令になっていたりするが，実現できる操作は同じである．これを基本として，手続き呼び出し命令や，割り込み，動作状態を操作するシステム制御命令が付加される．

MPUの種類によっては，整数と浮動小数点以外のデータ型をサポートしているものもある．それら新しいデータ型に対しては，専用の演算命令が用意される．たとえば，連続する整数データを文字列またはビット列とみなして，それらに特殊な処理を施す命令(文字列の転送，文字列の比較，文字やビットのサーチ)が考えられる．

また，いくつかの命令機能を一つの命令で実現させるようにすると，命令数はどんどん増加していく．その極端な例がCISCであろう．

● 2オペランド形式と3オペランド形式

MPUの命令は，命令コード(オペコード)とオペランドからなる．このオペランドの個数が，命令セットアーキテクチャを特徴付ける一要素となっている．これには，オペランドを2個もつ**2オペランド形式**と，オペランドを3個もつ**3オペランド形式**がある．

一般的なデータ処理を考える場合，転送はソースオペランドとデスティネーションオペランドの二つが決まれば実現できる．しかし，演算には二つのソースオペランドと一つのデスティネーションオペランドが必要である．つまり，ソース1とソース2を演算して結果をデスティネーションに格納する．よって，オペランドの個数としては3個がもっとも自然であろう．

しかし，世の中のMPUは2オペランド形式を採用しているものも多い．この形式は，演算において片方のソースをデスティネーションと兼用する．また，片方のソースが破壊されるという意味で，3オペランド形式よりもプログラミングの自由度が低い．では，なぜ2オペランド形式が採用されるのかというと，それは命令長を節約できるからである．

● 命令長について

たとえば，アーキテクチャ的に32本のレジスタを使用できる場合を考えると，レジスタを指定するためにはオペランドに5ビット分の領域が必要である．一つの命令内でレジスタを指定する総ビット数は，2オペランド形式では5×2＝10ビット，3オペランド形式では5×3＝15ビットである．もし，命令長が固定されていると考えると，3オペランド形式では2オペランド形式に比べて5ビット分もオペコード指定に使えるビット数が減ってしまう．つまり，命令の種類が制限されてしまう．逆に考えると，同じ数の命令を実現するには，3オペランド形式は2オペランド形式よりも命令長が長くなるのだ．

一般的に，RISCは命令デコードのしやすさとの兼ね合いから32ビット固定長の命令を採用する場合が多い．しかし，x86に代表されるCISCは，バス速度が遅かった昔の名残で，命令コードを短時間に取り込む工夫，つまり，命令長を短くする工夫をしている．その一環が2オペランド形式である．さらにCISCでは，多様なアドレッシングモードを指定可能とするために，ただでさえ長くなりがちな命令長をバイト単位の可変長にすることで対応している．

● アキュムレータ形式/スタック形式

オペランド形式としては，2オペランド形式，3オペランド形式のほかに，演算可能なレジスタをアキュムレータ(特殊レジスタ)に限定する**アキュムレータ形式**，演算をスタック上で行う**スタック形式**がある．

アキュムレータ形式は，演算器に直結するレジスタをアキュムレータに限定する．オペランドの一つがアキュムレータであることがわかっているので，その分だけ命令コードを短くできる利点がある．これは，トランジスタの集積規模が小さく，すべてのレジスタを演算器に接続できなかった昔のMPU，たとえば，8080やZ80によく見られる．

演算をスタック上でしか行わない点で，スタック形式はアキュムレータ形式の特殊なものとみなすこともできる．しかし，アキュムレータが，通常は一つしかないのに対して，スタックは理論上無限の個数があり，複数の中間結果を同時に格納できるという点で，式の計算を実現するのに便利である．

2　CISCの命令セット

● 複雑な命令セットのコンピュータ＝CISC

当然のことながらRISC誕生以前は，いわゆるCISCしかないわけで，CISC命令セットがコンピュータの命令セットの原点である．

CISCの命令セットは，一部に簡単な処理を行う命令もあるが，大半は複雑な処理を行う命令の集合であ

表1　V60/V70の命令の種類

●転送 ●整数演算 ●比較 ●論理演算 ●シフト/ローテート ●実効アドレス計算 ●単一ビット操作 ●ビットフィールド(挿入，抽出，比較) ●ビットストリング転送 ●文字ストリング転送 ●10進演算 ●浮動小数点演算 ●手続き呼び出し ●分岐 ●PSW (Program Status Word)操作 ●MMU制御 ●入出力 ●タスク制御 ●アトミック(不可分)命令

る．複雑とは，一つの命令で多くの処理を行うためである．これはメモリのアクセス時間が遅かった時代にコンピュータの実行性能を高めるために行った自然な選択である．プログラミング言語のコンパイラやインタプリタで行う処理を少ない命令で効率的に実現したり，OSの操作を効率的に行ったりするための工夫が盛り込まれている．

その特徴をV60/V70(NEC)とTRONチップで見ていこう．これらのMPUの命令セットが完成した時期はCISCの後期に属し，その意味からも，ほかのCISCの命令セットの「いいとこ取り」であり，究極のCISCともいえるからだ．

● V60/V70の命令セットの特徴

V60/V70の命令の種類を**表1**に示す．命令長は1バイトから22バイトまで存在し，2オペランド方式である．そして，ソースとデスティネーションは21種のアドレッシングモードを独立して指定できる．これをV60/V70では「対称性」と呼んでいる．なお，命令長を短縮するために，片方のオペランドがレジスタの場合は短縮型の命令形式が用意されている．また，データ型は次に示す14種で，それぞれのデータ型に関してすべての演算(意味がある場合)が定義されている．これを「直交性」と呼んでいる．

- 整数(バイト，ハーフワード，ワード，ダブルワード)
- ポインタ
- ビット
- ビットフィールド
- ビットストリング
- 10進数(パック，アンパック)
- 文字ストリング(バイト，ハーフワード)
- 浮動小数点データ(単精度，倍精度)

このように，V60/V70の命令セットの特徴は対称性，直交性に優れていることである．これはプログラムの書きやすさはもちろんだが，コンパイラの作成を容易にするという意図がある．

● 命令の特徴

V60/V70で特色のある命令を見ていこう．その項目を検証することで，CISCの命令セットがどのような項目を重要と考えていたかが推測できる．

▶ 手続き呼び出し関連

手続き呼び出し命令は，高級言語のコンパイラを実現するための要である．Cコンパイラにおいて，手続き呼び出しは一般に次のようなシーケンスをとる．

- 引き数をスタックに積む(PUSH命令)
- 手続きを呼び出す(CALL命令)
- ローカル変数のためのスタックフレームを作成する(PREPARE命令)
- レジスタ変数に使用するレジスタを一括して退避する(PUSHM命令)
- 手続きの実行
- 退避したレジスタを一括して回復する(POPM命令)
- スタックフレームを解放する(DISPOSE命令)
- 手続きから復帰する(RET命令)
- 引き数領域を開放する
  (POP命令またはADDによるスタックの補正)

このようにV60/V70では，それぞれの処理に対応する専用命令が用意されている．V60/V70の手続き呼び出しで特徴的なのは，アーギュメント(引き数)ポインタという概念である．これは引き数を参照するためのベースレジスタであり，CALL命令によって値が設定される．アーギュメントポインタは，Cコンパイラではcall命令実行時のスタックのトップだが，FORTRANやCOBOLでは別の場所になる．それらに対処したわけである．ローカル変数に関しては，ほかのアーキテクチャと同じくフレームポインタをベースとして参照する．またRET命令は，オペランドの値でスタックポインタを補正することもできる．つまり，スタックにある引き数領域をRET命令実行時に解放することもできるわけだ．Cコンパイラでは呼び出し側で引き数領域を解放するので，これはPASCALコンパイラ用である．

なお，これらの手続き呼び出しシーケンスは，VAXのそれに非常に強い影響を受けていることを付け加え

ておく．

▶ ビットストリング操作/ビットフィールド操作

この命令はビットマップグラフィックのデータ処理に用いる．この命令により，メモリ中の任意のビット位置から任意のビット長のビット列どうしにNOT，AND，OR，XOR，AND-not，OR-not，XOR-notなどの論理演算を施すBitBlt処理を行える．ビットストリング操作のうち，ビット列の連続する0または1を計数する命令は，現在でもFAX処理や画像の圧縮伸張に応用できる．

ビットストリング命令に似た命令にビットフィールド命令がある．これはメモリの任意の位置から指定したビット長のフィールドを抽出/比較したり，メモリの任意の位置から指定したビット長のフィールドにデータを挿入したりできる．この命令も画像の圧縮伸張に使用可能だ．

▶ 文字ストリング操作

C言語でいうところのstrcpy，strcmp，strlenなどのライブラリ関数の機能を1命令で実行できるものだ．転送の単位は8ビットと16ビットがあり，漢字コードの転送にも考慮している．ソースとデスティネーションの文字ストリングがオーバーラップする場合も正常な転送ができるように，逆方向から転送する命令もある．これらの命令は大型計算機のACOSのデータ転送命令を参考にしたといわれている．

▶ 10進演算

COBOLなどでの使用を考慮し，10進数の文字コードを直接加減算する命令がある．BCD形式(パック型)の10進数も演算できる．現在は不明だが，以前，世界でもっとも多く使われているコンピュータ言語はCOBOLだった．10進演算命令は，COBOLにおける数値処理を高速に処理するためのものである．

▶ MMU制御

ATE(Area Table Entry)や，PTE(Page Table Entry)といったアドレス変換テーブルの内容を，その各エントリに関連する仮想アドレスによる指定で直接リード/ライト可能なUPDATE，GETATE，UPDPTE，GETPTE命令や，仮想アドレスと対になる物理アドレスを得るGETRA命令などがある．また，各実行レベルからのアクセスの可否を判断するCHKAR/CHKAW/CHKAE命令，実行レベルを変更するCHLVL命令がある．TLBの操作に関しては，指定した仮想アドレスにヒットするエントリを無効化するCLRTLB命令と，すべてのエントリを無効化するCLRTLBA命令がある．TLB内容の入れ替えは自動的に行われるため，TLBの内容を直接操作する命令はない．

▶ コンテキスト切り替え

マルチタスク環境下でのタスク切り替えを1命令で実行する，コンテキスト切り替え命令(LDTASK/STTASK)がある．この命令は，V60/V70のレジスタセットや仮想記憶情報を選択的にメモリ中にあるタスク制御ブロックの内容と入れ替えることができる．

▶ アトミック命令

マルチプロセッサ環境でのセマフォを実現するためのテストアンドセット命令(TASI)と，コンペアアンドスワップ命令(CAXI)がある．これらの命令はバスをロックして操作を行うアトミック命令である．

▶ 非同期トラップ

これは命令ではなく，OSの機能をサポートするしくみである．非同期というのは，トラップが発生する条件があってもただちに例外処理に移行するのではなく，あと(RETI命令の実行時)まで遅延させることを意味する．つまり，条件の成立と発生が同時でないことを示す．V60/V70では，OSのための非同期システムトラップとユーザータスクで使用できる非同期タスクトラップが提供される．

● V80での高速化項目

V60/V70の後継機種であるV80の命令セットは，基本的にV60/V70と同一である．機能的にはV60で完成していると考えられたからだ．V60/V70の命令セットを見ると，コンパイラの作成しやすさ，OSの書きやすさを第一に考えているのがわかる．V80ではこれをさらに高速化することに注力している．具体的には，次に示すような項目である．

- 基本命令のハードワイヤードロジック化
- SP(Stack Pointer)のフォワーディング
- CALL/RETの高速化，RETIの高速化
- 文字ストリング操作のハードウェア化
- ビットフィールド操作の高速化
- ビットストリング操作の高速化
- TLB入れ替えのハードウェア化
- FPUの高速化，乗算器
- 分岐予測機能の採用
- キャッシュの採用
- アトミック命令の追加(ADDI/SUBI/ANDI/ORI)

これらの機能の導入により，**表2**に示すような性能向上が得られたという．現在のMPUの実行クロック数からみればかなり低性能であるが，当時としてはか

表2　V80での高速化の実際(実行クロック数)

機　能	命令，条件など	V70	V80
転送	MOV.W　reg,mem	4	2
	MOV.W　mem,reg	4	2
整数演算	ADD.W　reg,mem	2	2
	ADD.W　mem,reg	4	2
	ADD.W　mem,mem	8	4
整数乗算	MUL.W	23	9
整数除算	DIV.W	43	39
シフト	SHA.W	17	3
分岐	Branch Taken	11	2
	Not Taken	4	4
手続き関連	CALL & RET	44	21
ビット フィールド	EXTBFZ	30	10
	INSBFZ	28	10
文字列操作	MOVCU.B(n bytes)	$30+5n$	$19+1.25n$
浮動小数点 (単精度)	ADDF.S	120	36
	MULF.S	116	44
	DIVF.S	137	75
浮動小数点 (倍精度)	ADDF.L	78	75
	MULF.L	270	110
	DIVF.L	590	553
割り込み復帰	RETIS	80	22
コンテキスト 切り替え	LDTASK(44words)	347	157
	STTASK(44words)	200	121
TLBミス処理	異なるエリア	58	11
	同一エリア	58	6
割り込み応答	ハンドラ実行まで	165	27

なり高速だった．

● TRONチップの命令セットの種類

TRONチップの命令セットで提供される命令の種類を表3に示す．これはどう贔屓目に見てもV60/V70の命令セットの2番煎じの感を免れない．基本的な命令セットはほぼ同じで，あえて新規性を見出すとすればキュー(待ち行列)操作命令くらいだろうか．これを好意的に考えれば「目指すところは誰でも同じ」といえるだろう．

V60/V70のほうがTRONチップよりも歴史が古いのだが，V60/V70は歴史の中に埋もれ，官民一体となった研究の強みなのか，TRONチップは日本オリジナルのMPUとして今だに言及されることがある．

TRONチップの命令セットの特徴は基本命令の高速実行と命令の対称性の実現ということに尽きる．高速実行とは短い命令長のことを指し，命令フェッチが高速に行えることを意味する(命令キャッシュのない状況では)．算術演算命令についていえば，V60/V70では最小の命令長が3バイトだったのに対し，TRONでは2バイトである．ただし，これは汎用レジスタの本数がV60/V70では32本であるが，TRONでは16本なので命令の符号化が少ないビット数で行えることも一因であろう．

表3　TRONチップの命令の種類

- 転送命令
- 比較命令
- 算術演算命令
- 論理演算命令
- シフト命令
- ビット操作命令
- 固定長ビットフィールド操作命令
- 任意長ビットフィールド操作命令
- 10進演算命令
- ストリング命令
- キュー操作命令
- 分岐命令
- マルチプロセッサ命令(アトミック命令)
- 制御空間，物理空間操作命令
- OS関連命令
- MMU関連命令

このように，短い命令長を実現しながら命令の対称性も両立させている．そして，この対称性こそがTRONチップの命令セットの本質である．TRONチップが発表された時代はソフトウェア危機が真剣に議論された時代であり，ソフトウェアの生産性を高める命令セットが最善とされていた．つまり，それはソフトウェアの作りやすい命令セットであり，取りも直さず対称性のよい命令セットである．この考えはコンパイラを作りやすい命令セットへと行き着く．TRONチップの提唱者である坂村健氏による命令セットの解説を，参考文献3)より以下に引用する．

「高級言語でプログラミングを行う場合には，プログラマから直接プロセッサのアーキテクチャが見えるわけではないので，質のよいコンパイラさえできれば，アーキテクチャがどうなっていてもかまわないという考えがある．しかし，プロセッサのアーキテクチャが悪いと，質のよいコンパイラを作るのが非常に難しくなり，実用的なコンパイラができるまで何年もかかってしまう場合がある．したがって，コンパイラが作りやすいことは，チップを普及させるための重要な要素である．」

対称性が良く短い命令長を実現するために，TRONチップでは，命令の対称性が良く機能の高い「一般形」の命令形式と，機能は制限されるが命令長の短い「短縮形」の命令形式の2種類を用意している．MOV(転送)命令やCMP(比較)命令のように出現頻度の高い命令に対してはより多くの種類の短縮形が用意されている．この考え方はV60/V70も同じである．しかし，

TRONチップではこのほかに命令の対称性を高めるために，「多段間接アドレッシングモード」と「異種サイズ間の演算機能」を用意している．一般にアドレッシングは，スケーリング，加算，間接参照の組み合わせであるが，TRONチップはこの3種の演算を自由に複数回組み合わせることができる(実装上の回数制限はある)．坂村氏は，上述の文献でこの理由を次のように述べている．

「多段間接モードは，非常に汎用性の高い間接アドレシングの機能である．(中略)この機能は，AI応用やモジュール化プログラミング等に特に効果がある．従来のプロセッサの場合は，間接参照の機能があったとしても，インデックス用レジスタを加算するのは間接参照後に限られるとか，2個のインデックス用レジスタを加算することはできないとかといった制限が多く，コンパイラがそれを利用できる場合は限られていた．」注1

「一般に，アドレス演算やレジスタ上での演算はそのプロセッサの基本サイズ(レジスタのサイズ)で行われるのが普通であるが，メモリにデータを格納するときには，データの範囲に合わせた最小限のサイズを使用することが多い．したがって，データのサイズ変換を行う頻度はかなり高い．TRONチップの異種サイズ間演算機能を利用すると，整数データのサイズ変換と演算を同時に行えるため，効率のよいプログラミングが可能である．」注2

● TRONチップの命令セットの特徴

さて，TRONチップの命令セットの特徴は(坂村健氏が提唱するリアルタイムOSである)ITRON，BTRONをサポートするために便利な機能を命令として提供することである．さすがにTRONというOSを動かすために提唱された命令セットである．これらについて以下に説明する．

▶ 分岐命令

これはOS用の命令ではないが，特に取り上げておく．TRONチップではループ処理の終端で現れる命令列の加算/比較/分岐(ACB命令)，減算/比較/分岐(SCB命令)といった命令列を1命令で提供する．これは一般的な減算&分岐命令(Decrement and Branch)を拡張したものであるが，いかにもCISCという感じである．多分，当時のアーキテクチャ設計者は誰でも導入を考えたはずだ．

▶ コンテキスト切り替え命令

これはV60/V70におけるコンテキスト切り替え命令と同じである．ITRONにおけるタスク切り替えを高速に行うため，コンテキストのロード(LDCTX)/ストア(STCTX)命令を提供する．

▶ キュー操作命令

ITRONにおけるタスクのレディーキュー管理のための，双方向リンクキューに対する挿入(QINS)，削除(QDEL)，サーチ(QSCH)命令がある．この命令は割り込みが入ってからそれを処理するタスクが起動するまでの時間を高速化し，リアルタイム応答性を向上させる．

▶ 可変長ビットフィールド命令

これはV60/V70におけるビットストリング操作命令と同じである．ビットマップディスプレイでのウィンドウ操作のための命令だ．任意長のビットフィールド間でのビットマップ演算命令(AND，ORなど16種類)，ビットマップ転送(BVCPY)，同一パターンとの演算(BVMAT)，0または1のサーチ(BVSCH)のための命令を用意している．

▶ ストリング操作命令

これはV60/V70における文字ストリング操作命令と同じである．文字(テキスト)処理を高速に実行するためのストリングのコピー(SMOV)，比較(SCMP)，サーチ(SSCH)，フィル(SSTR)がある．BTRON向けの命令セットである．

▶ マルチプロセッサ命令

バスをロックした状態で，指定したアドレス中の指定ビットをセット(BSETI)，クリア(BCLRI)，比較とストア(CSI)を実行する．

▶ MMU関連

アドレス変換テーブルのエントリの内容の更新(LDATE)，参照(STATE)，TLBのパージ(PTLB)，仮想アドレスの論理アドレスへの変換(MOVPA)，4レベルの実行レベルから指定のアドレスをアクセスできるかどうかのチェック(ACS)を行う命令がある．TRONチップにおいてもTLB入れ替えは自動的に処理され

注1：筆者意見…説明が的を射ていない気がする．メモリ間接の意義は合目的なものではなく，ポインタ処理の高速化にある．しかしメモリを1回以上参照すると，パイプラインが乱れるのであまり効果はない．とはいえ，命令コード(バイト数)の節約にはなる．

注2：筆者意見…この機能はメモリと直接演算する場合は便利かもしれない．しかしたいていの最適化コンパイラは，データをレジスタに引き上げてから演算するので無意味ではないか．ロード/ストアアーキテクチャの対極にあるような機能だと思われる．当時のコンパイラの最適化技術を考えれば，当然の帰結だったかもしれないが…

表4　V60/V70/V80とTRONチップの性能比較

MPU	性能	開発元	発表時期
V60	3.5MIPS@16MHz	(NEC)	1986年
V70	6.6MIPS@20MHz	(NEC)	1987年
V80	16.5MIPS@33MHz	(NEC)	1989年
Gmicro/100	5MIPS@20MHz	(三菱電機)	1989年
Gmicro/200	6MIPS@20MHz	(日立製作所)	1988年
TX1	5MIPS@25MHz	(東芝)	1988年
Gmicro/300	17MIPS@25MHz	(富士通)	1989年
Gmicro/400	???	(三菱電機)	????
Gmicro/500	132MIPS@66MHz	(日立製作所)	1993年

るため，TLBの内容を直接操作する命令は用意されていない．

● **実チップでの実装**

当時の文献を読むと，TRONチップの命令セットを実チップに実装する場合，V80の場合とは異なったアプローチが採られていることに気付く．V80では性能のキーポイントとなる命令や操作を高速化することでプログラム自体の性能を向上させようとしている．それに対し，TRONチップはパイプライン処理をいかに効率的に行うかに注力している．チップの実装としてはTRONチップの方式のほうが王道という気もするが，命令処理の実行クロック数が一定にならないCISCにおいて，本当に効率的なパイプライン処理を実現できたか否かは疑問である．

TRONチップの後期を飾ったGmicro/100/300/500で採用された高速化手法を，とりあえず列挙しておく．Gmicro/300のアプローチはV80のそれに近いかもしれない．**表4**にV60/V70/V80と各TRONチップの性能比較(公称値)を示す．Gmicro/500は別格としても，V80やGmicro/300での高速化はそれなりに成功しているといえよう．Gmicro/500に関しては，Pentium(無印)と同時期に登場し，同性能の性能を得ることができたと，少し前の坂村氏の論文に書いてあったが，Gmicro/500は実際に出荷されたのだろうか．日立製作所と三菱電機が製造して1994年(？)に出荷予定という新聞記事は見たことあるが…

▶**Gmicro/100における高速化**

- 分岐予測
- 分岐バッファ(分岐先命令のキャッシュ)

▶**Gmicro/300における高速化**

- 基本命令の1クロック実行
- 10進命令の高速化
- キャッシュ
- ストアバッファ

▶**Gmicro/500における高速化**

- スーパースカラ
- 分岐命令キャッシュ(無条件分岐のみ)
- 分岐復帰バッファ(8レベル)
- ストアバッファ

3　崩れた神話――RISCへ至る道

● **直交性に優れた命令を用意したが……**

コンパイラに優しいCISCの命令セットは，良質のコンパイラの登場を約束するはずだった．しかし，現実はその思惑どおりには進まなかった．多種多様な命令とアドレッシングモードがあればコンパイラを作りやすいのは確かである．しかし，コンパイラが生成する命令コードの性能という観点から見ると，パイプラインを有効活用できないような複雑な命令は不利だということがわかってきた．

CISCの時代にはまだバス速度が遅く，複雑な1命令がよいか，単純な複数命令がよいかということは一概にはいえなかった．しかし，内蔵キャッシュが一般的になると，単純な複数命令のほうが有利となった．また，パイプライン処理を前提とすると，一つの命令で多くの処理を行うよりも，単純な命令に分解して実行するほうがスループットも高い．さらに，コンパイラの最適化技術が進むと，ロード/ストア以外ではメモリ参照を行わなくなる．つまり，命令のオペランドはレジスタだけで事足りる．そして，パイプラインを乱すメモリ間接アドレッシングはほとんど使用しなくなったのだ．

● **単純な命令に置き換えて性能向上**

バス速度がある程度速いという条件で，CISCの専用命令を単純な命令に置き換えることにより，パイプラインがスムーズに流れるようになり，結果として性能が向上する例をいくつか示す．

▶**V60/V70のスタック操作命令**

時代の経過とともに，この置き換えと同様な処理は，x86のコンパイラでも積極的に採用されるようになった．つまり，ENTER/LEAVE命令はPUSH/POPとADD/SUB命令に置き換えられ，PUSHA/POPAが使用されることはほとんどなかった(**リスト1**)．

▶ **一般的な手続き呼び出し**

この置き換えの発想は，RISCのJAL(Jump And Link)命令にある．これは，手続き呼び出しに(でき

リスト1　V60/V70のスタック操作命令

```
    PUSHM 0m<r1-r3>

/* 命令置き換えその1
(PUSHMは余分な前処理と後処理が必要なので遅いためそれぞれPUSHする) */
    PUSH    r1
    PUSH    r2
    PUSH    r3

/* 命令置き換えその2
(PUSHの連続はスタックポインタがレジスタハザードになるので,
spをインデックスにしてストア) */
    MOVE.W r1,-4[sp]
    MOVE.W r2,-8[sp]
    MOVE.W r3,-12[sp]
    ADD.W  #-12,sp
```

(a) Push Multiple(複数レジスタの同時プッシュ)

```
    PREPARE #12

/* 命令置き換え */
    MOVE.W fp,-4[sp]
    ADD.W  #-4,sp
    MOVE.W sp,fp
    ADD.W  #-12,sp
```

(b) Prepare Stack Frame
(スタックフレームの生成)

```
    DISPOSE

/* 命令置き換え */
    MOVE.W 0[fp],fp
    ADD.W  #4,sp
```

(c) Dispose Stack Frame
(スタックフレームの破棄)

リスト2　一般的な手続き呼び出し

```
  JSR   target

/* 命令置き換え */
    LoadAddr next,ra /* 戻りアドレスをレジスタ(ra)に入れて */
    JUMP     target  /* targetへジャンプ */
next:
```

(a) Jump to Subroutine

```
  RSR

/* 命令置き換え */
  JR       ra        /* レジスタ間接ジャンプ */
```

(b) Return from Subroutine

るだけ)スタックを使わないというものである．もちろん引き数はレジスタ渡しする．この場合，従来の手続き呼び出し命令はジャンプ命令を用いて**リスト2**のように置き換えられる．戻りアドレスはレジスタに格納する．

さすがにここまで割り切ったコンパイラは少なかったが(せっかく用意されているCALL命令やJSR命令を使わないのは心苦しかったのだろう)，引き数のレジスタ渡しは積極的に行われるようになった．RISCコンパイラの技術を受けてCISCのコンパイラもそれなりに進化したのである．

● TRONでは逆の主張もあったが…

TRONチップのWebサイトで次のようなプログラム例を見つけたので紹介しておく．ここでの趣旨は，いわゆるCISC命令は単純なRISC命令で置き換えても性能は向上しないというものだ．

実例がキャッシュを搭載していない(分岐バッファを命令キャッシュに割り当てることはできるが)Gmicro/100での話なので，Gmicro/300やGmicro/500では事情は異なると思うが，一応掲載されたままの性能値を紹介しておく．

▶ ダブルリンクトキューの挿入命令

リアルタイムOSにおいて，QINS命令はタスク切り替え時のレディーキューの操作に使用される．TRONアーキテクチャでは，これを1命令で実現でき，12バイト長で18クロックの実行時間である．これを単純な命令で置き換えると**リスト3**のようになる．これには36バイトが必要で，実行時間は26クロックである．これでは各命令が1クロック実行になると性能が逆転するのではないだろうか．

リスト3　ダブルリンクトキューの挿入命令

```
    QINS     @(R1,FOR)<@(RDQ_TBL,R2*8)

/* 命令置き換え */
    MOV      @(RDQ_TBL+4,R2*8),Rn
    MOV      R1,@(RDQ_TBL+4,R2*8)
    MOVA     @(RDQ_TBL,R2*8),@(R1,FOR)
    MOV      Rn,@(R1,BACK)
    MOV      R1,@(Rn,FOR)
```

▶ 可変長ビットフィールド内のビット検索

これは最高の優先順位を持つタスクを見つけるために，リアルタイムOSのタスクスケジューラで使われる．この操作はTRONアーキテクチャのビットフィールド操作命令を使用して**リスト4**のように記述できる．これは14バイトのコード長で，実行に62クロックかかる．

これを単純な命令で置き換えると，78バイトのコード長で実行に244クロックかかる．

このプログラムはうまく書けば2回目のループは不要であり，メモリ参照の回数を削減できる．最後の2

リスト4　可変長ビットフィールド内のビット検索

```
    MOVA      @RBQ_BIT,R0
    MOV       #0,R1
    MOV       #272,R2
    BVSCH/F/1

/* 命令置き換え */
    MOVA          @RDQ_BIT,R0
                        MOV         #0,R1
                        MOV         #0,R2
                        MOV         #7,R3
                        MOV         #0,R4
      SRCH1:            CMP.W       #0,@R0
                        BNE         FOUND1
                        ADD         #4,R0
                        ACB         #1,R1,#8,SRCH1
      FOUND1:
      SRCH2:            CMP.B       #0,@R0
                        BNE         FOUND2
                        ADD         #1,R0
                        ACB         #1,R2,#3,SRCH2
      FOUND2:           MOV.B       @R0,R4
                        MOV.W       #H'FE,R5
      SRCH3:            AND.B       R5,R4
                        BEQ         FOUND3
                        SHL.B       #1,R5
                        SCB         #1,R3,#0,SRCH3
      FOUND3:           SHL         #5,R1
                        SHL         #3,R2
                        ADD         R2,R1
                        ADD         R3,R1
```

個のシフト命令も不要になる．実際は244クロックより短い時間で実行可能と思われる．

▶ 結局は効果がないのでは…

上の二つの例は，複雑な命令の有利性が確かだとしても，OS内処理の高速化の話である．たしかに，リアルタイムOSでは重要かもしれない．しかし，これがアプリケーションプログラムの性能向上に直結するとは考えにくい．

また，これらの命令が出現する頻度はほかの命令に比べると非常に低いものであろう．たとえそうでないとしても，さらにプロセッサのRISC化が進んで命令実行が1クロック処理になり，スーパースカラなどが導入されると性能差はなくなってくると思われる(3倍程度の性能にはなる？)．

● 単純命令を高速に実行

CISC命令の高速化の過程で生まれてきた考え方の一つは，使用頻度の高い命令を高速に実行することである．性能にクリティカルな命令を高速に実行するためにワイヤードロジック化したり，専用のハードウェアを導入することが考えられた．しかし，複数の処理を1度に実行する命令を高速に実現するためには大規模な専用のハードウェアが必要になり，チップ面積も大きくなってしまう．

この状況に新たな道を見出したのがRISCの研究である．統計をとると，単純な命令ほど使用頻度が高く，かつ性能に効いてくる．単純な命令を高速化するためには大規模なハードウェアは不要であり，また少し単純なハードウェアが全体の性能に効くのである．そして単純な構造のため動作周波数も上げやすい．

4 誕生初期のRISC

● ロード/ストアアーキテクチャと単純なアドレッシングモード

初期のRISCとして有名なものはIBM801，バークレーRISC I，スタンフォードMIPSである．これらの命令セットの特徴を以下に説明する．しかし，その前に，いわゆるCISCとその命令の特徴を**表5**に示しておく．一見すると，RISCの命令数は「縮小命令セットコンピュータ」の名のとおり，CISCの命令数よりもかなり少ない．しかし，これは，CISCでは同じ種類の命令であっても処理するデータサイズによって異なる命令とみなされるためである．そのため命令数が多く見える．

一方RISCでは，ロード/ストア以外は，レジスタ間で演算が行われるため，演算に関してデータサイズという概念がない．後述の命令セットの具体例を見ると，CISCとRISCの命令数(種類)には大差がないことがわかる．RISCをRISCたらしめている特徴は，

表5　CISCとRISCの命令セットの比較

	CISC			RISC		
	IBM370	VAX11/780	V60	IBM801	RISC I	MIPS
発表時期	1973	1978	1986	1980	1981	1983
命令数	208	303	273	120	23	55
マイクロコード量	54K	61K	23K?	0	0	0
命令長(バイト)	2～6	2～57	1～22	4	4	4
演算対象	reg-reg	reg-reg	reg-reg	reg-reg	reg-reg	reg-reg
	reg-mem	reg-mem	reg-mem			
	mem-mem	mem-mem	mem-mem			

ロード/ストアアーキテクチャと単純なアドレッシングモードであるといえよう(表6).

● パーシャル(部分)レジスタライト

CISCからRISCへの移行の間に，多くのユーザーが忘れてしまった特徴に**パーシャルレジスタライト**がある．これは，たとえばレジスタが32ビット長の場合，8ビットまたは16ビットの演算に対して，それぞれのビット長に対応する部分だけしか変更されないというものである．つまり，8ビット演算なら，レジスタの上位のビット31～8は変更されない．

これはx86のHIレジスタ，LOレジスタへの独立アクセスあたりにルーツがあるように思える．x86と同程度に古いアーキテクチャであるMC68000も，同様の特徴を有していた．

このパーシャルレジスタライトの概念をくつがえしたのがRISCアーキテクチャである．演算自体にデータ長というものがなく，必ずレジスタ全体が変更される．データ長という概念をもっているのはロード/ストア命令のみである．

ロード命令に関しては，レジスタ長にゼロ拡張/符号拡張されて格納される．つまりレジスタ全体が変更される．ストア命令に関しては，メモリに対して部分ライトされる．これはCISCと同じである．

現在のMPUではx86以外にはパーシャルレジスタライトの特徴は見受けられない．x86でさえ，パイプラインがストールするので，パーシャルレジスタライトの使用は推奨されていない．

● 条件フラグと条件分岐

CISCのユーザーがRISCのアーキテクチャを最初に見て奇異に思うのは，条件フラグが存在しないということだろう．CISCでは，ほとんどすべての命令で条件フラグが変化する．そして，条件分岐は最終的な条件フラグを参照して分岐するか否かを決定する．

一方RISCには，原則として条件フラグがない．条件分岐はレジスタの値が「0であるか」,「正であるか」,「負であるか」，あるいは二つのレジスタの値が「等しいか」,「等しくないか」という簡単なテストで，分岐するか否かを決定する．

RISCで条件フラグをなくした理由は，「フラグハザード」というパイプラインハザードをなくしてパイプライン処理をスムーズに行うためであろう．フラグハザードとは，条件フラグが確定するまで条件分岐命令の分岐先フェッチができずに，パイプラインが停止する状況を指す．

表6 RISCの特徴

● 命令の1サイクル実行
● メモリインターフェースは単純なロード/ストアのみ
● レジスタ間での演算
● 単純な形式の固定長命令
● 単純なアドレッシングモード
● 多数の汎用レジスタまたはレジスタウィンドウ
● 遅延分岐
● キャッシュ
● 高級言語コンパイラへの依存

CISCにおいて，条件分岐が参照する条件フラグは，原則的に条件分岐命令の直前の命令で確定する．その命令が加減算のように1クロックで実行できるものなら，それほど害はない．しかし，乗除算命令のように演算に数クロックを要する場合は，その分だけパイプラインがストールする．また，条件分岐命令の前方にある命令列は，条件フラグの値が変わってしまうので，気楽に並び替えることはできない．

RISCにおいて，命令の並び替えは，レジスタの依存性をなくすために日常茶飯事である．この目的のために条件フラグは邪魔になる．単にレジスタの値を参照するだけであれば，レジスタの値はフォワーディングされることもあり，レジスタの値が確定するまでの間のストールを最小限に抑えることができる．もし，RISCで条件フラグを採用するとすれば，その値を予測しフォワーディングすることが必要になり，ハードウェア量の増加を招く．このため，RISCでは条件フラグを用いないことが多い．

● バークレーRISC I/RISC II

カリフォルニア大学バークレー校のRISCの研究は，高級言語コンパイラが複雑な命令を有効に使えないことに着目することから始まった．プログラム実行時の命令の出現頻度，アドレッシングモード，変数の使われ方などの統計を採って，新しい命令セット設計の指針とした．この研究結果は，同大学のPattersonとDitzelによる初めてのRISCに関する論文『The Case for the Reduced Instruction Set Computer』として1980年に発表された．

この論文は，シングルチップコンピュータにとって最適なアーキテクチャはRISCであると主張し，次のような利点があると指摘した．

▶ チップサイズの縮小

単純なプロセッサなら，少ないトランジスタ数で設計できる．このためCISCに比べて相対的に多くの機能を集積できる．さらに空いた面積を使って，キャッ

表7　RISC IIの命令セット

整数算術命令	
ADD	加算
SUB	減算
SUBI	減算（逆方向）
S	シフト
論理演算命令	
AND	論理積
OR	論理和
XOR	排他的論理和
ロード/ストア命令	
LDR	ロード
LDX	インデックス付きロード
STR	ストア
STX	インデックス付きストア
フロー制御命令	
JMPX	条件ジャンプ（インデックス付き）
JMPR	条件ジャンプ
CALLX	条件コール（インデックス付き）
CALL	条件コール
RET	条件リターン
RETI	割り込みからの復帰
CALLI	割り込みハンドラをコール
特殊命令	
LDHI	レジスタの上位に値を設定
GETLPC	PCを得る
GETPSW	PSWを得る
PUTPSW	PSWを変更する

表8　MIPSのユーザーレベルの命令セット

整数算術命令	
ADD	加算
SUB	減算
SUBR	減算（逆方向）
IC	バイト挿入
XC	バイト抽出
RLC	レジスタ対のローテート
ROL	左ローテート
S	シフト
MEMSETUP	乗算準備
MSTEP	乗算の1ステップ（2ビット単位）
UMEND	符号なし乗算終了
DSTEP	除算の1ステップ
SET	条件のテスト結果をセット
論理演算命令	
AND	論理積
OR	論理和
XOR	排他的論理和
ロード/ストア命令	
LD	ロード
ST	ストア
MOV	即値またはレジスタの転送
フロー制御命令	
BRA	分岐
JMP	ジャンプ
TRAP	トラップ

シュやMMU，FPUなどを1チップに内蔵できる．

▶ **開発期間の短縮**

単純なプロセッサは，設計にかける労力やコストが少なくて済む．

▶ **高性能化**

単純な論理ゆえ，高い動作周波数で実行できる．CISCと比べると同じ動作周波数でもIPC（1クロックで実行する平均命令数）が高いので相対的に高性能である．

この論文の理論をバークレー校の大学院生が実践したのが，RISC IとRISC IIである．RISC IIの命令セットを**表7**に示す．これらは当時のCISCよりも単純で，設計の労力も少なかったが，CISCに匹敵する性能を発揮していた．

かくしてバークレーRISCは，後のARMやSPARCアーキテクチャの基礎となるのである．またRISCという言葉は，バークレー校によって初めて使用された．

● スタンフォードMIPS

バークレーRISCと同時期，スタンフォード大学でもHennessyを中心にRISCの研究が行われていた．それがMIPSである．MIPSではハードウェアを簡単にするために，メモリアクセスはワードアクセスのみとし，バイト単位の操作が必要な場合は専用命令を使ってレジスタ上で処理するとした．また，汎用レジスタは16本だった．

MIPSは2レベルの命令セットをもつ．一つはユーザーレベルの命令で，これはより通常（CISC）に近い抽象的な命令である（**表8**）．この命令セットではレジスタの依存関係を考慮する必要はない．もう一つはマシンレベルの命令で，ALUピース，ロード/ストアピース，制御フローピース，特殊命令（手続き呼び出し）といった部分的な命令コードからなり，リオーガナイザはこれらのピースを組み合わせて実行可能な命令を生成する．このとき，レジスタの依存関係が考慮され，インターロックしなくて済むように命令の入れ替えを行う．

最初のMIPSプロセッサは実用的といえるものではなかった．しかし，スタンフォード大学の研究者たちは，その研究を推し進め，2Kバイトの内蔵命令キャッシュと256Kバイトの外付けユニファイドキャッシ

ユインターフェース，32本の汎用レジスタ，乗除算用の特殊レジスタ，ゼロレジスタ，5段パイプラインを特徴とする，MIPS-Xというプロセッサを設計した．

● 乗除算命令の処理

MIPSでは，ほとんどすべての命令を1クロックで処理することを目標としている．しかし当然例外もある．浮動小数点演算と一部のシステム制御命令を除けば，乗除算命令がそれにあたる．乗除算命令は，一般には，1クロックで処理できない．これを通常のパイプラインに組み込むと，パイプラインが乱れて性能低下につながる．

これを回避するため，MIPSでは乗除算を通常のパイプラインとは切り離し，ほかの演算と並列に処理するようになっている．このため，乗除算の出力(デスティネーションオペランド)として汎用レジスタとは別の専用レジスタを用意している．こうすることで汎用レジスタとの依存性を解消する．その専用レジスタが，HIレジスタとLOレジスタである．

32ビット×32ビットの乗算では積は64ビットであり，上位32ビットがHIレジスタに，下位32ビットがLOレジスタに格納される．一方，32ビット÷32ビットの除算では32ビットの商がLOレジスタに，32ビットの剰余がHIレジスタに格納される．プログラムでは，乗除算命令の後，数命令後に(乗除算の計算が終了したのを待って)，HIレジスタまたはLOレジスタから結果を汎用レジスタに転送することになる．こうすることにより，パイプライン処理に乱れを生じさせない．

● 非整列データ転送命令

MIPS命令セットには非常に特徴的な命令がある．それが**非整列データ転送命令**である．これは，メモリ内の非整列ワード(misaligned words)データを処理する．CISCでは普通にサポートされている機能であるが，たいていのRISCでは非整列なアドレスに対するワードアクセスは例外事象としてトラップを発生する．もちろん，バイト単位でデータを処理すれば，アドレスが整列されていようがいまいが関係ない．しかし，複数のバイトをひとまとめに転送したほうが処理速度が上がる．

MIPSはRISCでありながら，ワードに整列されていないアドレスに対するロード/ストアをサポートする．これはMIPS社の特許であり，かつて互換メーカーのLexra社と訴訟になっていた(現在は和解)のは，この機能の無断使用に関してである．具体的には，次の8命令が用意されている．

(1) LWL(Load Word Left)

ワード内の有効データをレジスタに左詰めする．ロードしたデータでレジスタを部分的に変更する．

(2) LWR(Load Word Right)

ワード内の有効データをレジスタに右詰めする．ロードしたデータでレジスタを部分的に変更する．

(3) SWL(Store Word Left)

レジスタ内に左詰めされたデータをワード内の有効領域にストアする．

(4) SWR(Store Word Right)

レジスタ内に右詰めされたデータをワード内の有効領域にストアする．

(5) LDL(Load DoubleWord Left)

ダブルワード内の有効データをレジスタに左詰めする．ロードしたデータでレジスタを部分的に変更する．

(6) LDR(Load DoubleWord Right)

ダブルワード内の有効データをレジスタに右詰めする．ロードしたデータでレジスタを部分的に変更する．

(7) SDL(Store DoubleWord Left)

レジスタ内に左詰めされたデータをダブルワード内の有効領域にストアする．

(8) SDR(Store DoubleWord Right)

レジスタ内に右詰めされたデータをダブルワード内の有効領域にストアする．

これらの命令を利用すれば，たとえば，R5(転送元アドレス)からR4(転送先アドレス)へのデータ転送をワード単位で行うためには，

```
loop: /* 終了条件は省略 */
  lwr   r8,0(r5)
  lwl   r8,3(r5)
  addiu r5, r5,4
  swr   r8,0(r4)
  swl   r8,3(r4)
  addiu r4, r4,4
  b     loop
```

のように記述できる(リトルエンディアンの場合)．R4とR5の値がワードに整列されている必要はない．

● ARM

ARMはバークレーRISCから，ロード/ストアアーキテクチャ，32ビット固定長の命令，3アドレス形式など，多くの特徴を採用した．しかし次の特徴は採用しなかった．

表9　ARMの命令セット

データ処理命令	
ADD	加算
ADC	キャリ付き加算
SUB	減算
SBC	キャリ付き減算
RSB	減算(逆方向)
RSC	キャリ付き減算(逆方向)
AND	論理積
ORR	論理和
EOR	排他的論理和
BIC	ビットクリア
MOV	転送
MVN	ビット反転して転送
CMP	比較
CMN	否定して比較
TST	ビットテスト
TEQ	一致テスト
MUL	乗算
MLA	積和
データ転送命令	
LDR	ロード
STR	ストア
LDMIA	多重レジスタロード
LDMIB	多重レジスタロード
LDMEA	多重レジスタロード
LDMED	多重レジスタロード
LDMDA	多重レジスタロード
LDMDB	多重レジスタロード
LDMFA	多重レジスタロード
LDMFD	多重レジスタロード
STMIA	多重レジスタストア
STMIB	多重レジスタストア
STMEA	多重レジスタストア
STMED	多重レジスタストア
STMDA	多重レジスタストア
STMDB	多重レジスタストア
STMFA	多重レジスタストア
STMFD	多重レジスタストア
フロー制御命令	
Bcc	条件分岐
BL	分岐とリンク(サブルーチンコール)
SWI	ソフトウェア割り込み

▶ レジスタウィンドウ

レジスタの占める面積が多いためコスト面で不採用になったが，その概念は割り込み時のシャドウレジスタに受け継がれている．

▶ 遅延分岐

例外処理の実装を複雑にするため．

▶ 全命令の1クロック実行

ロード/ストアを1クロックで実行するには，命令とデータを格納するメモリが分離されている必要があり，ARMが対象とするアプリケーションには高価すぎるため．

ARMは命令セットの使いやすさよりも，ハードウェアでの実装を簡単に行えることを目標としている．この意味で，ARMの命令セットは，RISCの指針を受け入れながらも，保守的(CISC的)であるといえる．これは単純なハードウェア構成でありながら命令のコード効率を引き上げようとしたためである．**表9**にARM(ARM2相当)の命令セットを示す．

ARM命令の特徴は，すべての命令で条件コードを設定できること，第2オペランドをシフトして演算できること，演算を条件実行できることなどである．これらの操作をうまく組み合わせれば最小限の命令数で目的の処理を達成することができる．しかし，条件コードがあるため，レジスタの依存性だけに注目して命令を並び替えると動作が異なる場合もあり，最適化コンパイラ泣かせである．

5　過渡期のRISC

1989年当時，RISCという触れ込みで市場に出ていたアーキテクチャの代表を挙げれば，i860(Intel)，88000(Motorola)，SPARC(Sun Microsystems)であろう．これらは**表6**に示すRISCの特徴を満たしていた．これらの特徴に加え，i860はグラフィックとベクタ処理の命令を，88000はビットフィールド命令を，SPARCはタグ付きデータ命令というCISC系の命令を有していた．このあたりに過渡期のアーキテクチャという性質を垣間見ることができる．

● i860

i860は，x86とは異なる新しいアーキテクチャを提供する目的で開発された．i386 + 80387の性能を上回る高性能を実現することができ，従来のスーパーコンピュータやミニコンが提供していた科学技術計算や各種のシミュレーションを，より小型で安価なシステムで実現できた．

i860は，現在でもDSPの代用品やRAID用のプロセッサとして生き残っている．

● 88000

88000というのは，MPUであるMC88100とキャッシュ，そしてMMUを内蔵するMC88200というチップの総称である．コードユニット，データユニット，整数ユニット，FPU(加減乗除と変換用の二つ)の計

五つのユニットがあり，各自がパイプラインで並行動作するという意味でスーパースカラのはしりである．

MC88100は比較結果を反映させる条件コードレジスタをもっていない．比較命令は，ほかの演算命令と同じように3オペランド命令で，比較結果をデスティネーションレジスタに格納する．条件分岐命令はこのレジスタの値に基づいて分岐する．この構成により，比較命令と条件分岐命令間の命令を自由にスケジューリング(入れ替え)できる．条件コードを使用しないこの方式は，MIPSをはじめとする多くのRISCで採用されている．

● SPARC

SPARCの仕様はオープンアーキテクチャとして，SPARC International社によって管理されている．SPARCにはいくつかのバージョンがあり，最新バージョンは9である．バージョン9は64ビットアーキテクチャであるが，(少し前の)典型的なSPARCチップは32ビットアーキテクチャのバージョン7または8の仕様に基づいている．

SPARCの最大の特徴は，レジスタウィンドウである．整数ユニットは32ビットの汎用レジスタを136個もっている．このうち8個はグローバルに参照できるが，残りは手続きごとに割り当てられ，引き数の授受を高速に行う．これがレジスタウィンドウで，一つのウィンドウは24個のレジスタからなる．内訳は，R24～R31が手続きの呼び出し元とオーバーラップする(引き数用)．R16～R23は手続き内でローカルに使用できる．R8～R15は手続きが呼び出す手続きとオーバーラップする．手続きの最初にレジスタウィンドウを切り替えることで，レジスタの値を退避することなく，レジスタを自由に使用できる．SPARCにおける手続き呼び出しのシーケンスは次のようになる．

- R24～R31に引き数をセットする
- CALL命令を実行する
- 呼び出された手続きはSAVE命令でレジスタウィンドウを切り替える
- 手続きを実行する
- RESTORE命令で元のレジスタウィンドウを回復する
- RET命令(JMPL命令の特殊形)で復帰する(実際はRET命令の遅延スロットにRESTORE命令を置く)

レジスタウィンドウに関しては，多くの利点があることがわかっている．一つ目は手続き呼び出しごとにレジスタの値の退避/回復を行う必要がない点である．

表10　i860の命令セット

ロード/ストア	6種
浮動小数点-整数レジスタ間転送	2種
整数算術演算	4種
シフト	4種
論理演算	8種
分岐・コール・トラップ	13種
浮動小数点乗算	6種
浮動小数点加算	12種
デュアルオペレーション	4種
長整数加減算	4種
グラフィック	10種
I/O	3種
システム制御	6種

二つ目は，高度なレジスタ割り付けを要求しないのでコンパイラがそれほど複雑にならない点である．

● i860/88000/SPARCアーキテクチャの比較

i860の命令セットに関しては，筆者の手元に詳細な資料がないので概要のみを**表10**に示す．また，88000とSPARCの命令セットを**表11**と**表12**に示す．

▶ 特殊命令

標準的な命令セットに加え，i860はグラフィック処理の命令，整数と浮動小数点演算の並列実行(VLIWの特色)，FPUをサポートする．グラフィック処理にはZバッファ操作，Phongシェーディング，ピクセル間演算がある．これらは陰面消去と3D投影に効果的である．しかし，これらの特徴はグラフィック処理以外では効果的でない．整数と浮動小数点の並行処理は浮動小数点演算が支配的なアプリケーション以外では効果がないし，専用のプリフィクスが必要なため，当時のコンパイラは並列実行のための専用コードを生成しなかった．アセンブラの助けが必要である．

88000のビットフィールド命令はビットフィールドの中に対してセット/クリア，挿入/抽出をサポートする．ビットフィールド命令は最近のRISCにおける命令拡張では流行になっている．つまり，先祖返り的な傾向が見られる．

SPARCのタグ演算はデータとポインタに異なるタグを付け，データやポインタに関する不正演算を検出する．これは，LISPやSmalltalkの実装(動的なエラーチェック)に非常に有利である．

▶ セマフォ

これらのMPUは，テストアンドセット操作を実現する命令をもち，セマフォをサポートする．i860にはロック/アンロックという命令の組がある．この間に

表11　MC88100の命令セット

整数算術命令	
ADD	加算
ADDU	符号なし加算
CMP	比較
DIV	除算
DIVU	符号なし除算
MUL	乗算
SUB	減算
SUBU	符号なし減算
浮動小数点算術命令	
FADD	浮動小数点加算
FCMP	浮動小数点比較
FDIV	浮動小数点除算
FLDCR	浮動小数点レジスタからのロード
FLT	整数→浮動小数点変換
FMUL	浮動小数点乗算
FSTCR	浮動小数点レジスタからのストア
FSUB	浮動小数点減算
FXCR	浮動小数点制御レジスタとの交換
INT	浮動小数点→整数変換
NINT	Nearest方向の整数変換
TRUNC	Zero方向の整数変換
論理演算命令	
AND	論理積
MASK	論理マスク即値
OR	論理和
XOR	排他的論理和
ビットフィールド操作命令	
CLR	ビットフィールドのクリア
EXT	ビットフィールドの抽出(符号拡張)
EXTU	ビットフィールドの抽出(ゼロ拡張)
FF0	0であるビットの検索
FF1	1であるビットの検索
MAK	ビットフィールドの生成
ROT	レジスタのローテート
SET	ビットフィールドのセット
メモリアクセス命令	
LD	ロード
LDA	アドレスのロード
LDCR	制御レジスタからのロード
ST	ストア
STCR	制御レジスタへのストア
XCR	制御レジスタとの交換
XMEM	レジスタとメモリの交換
フロー制御命令	
BB0	ビットクリア時に分岐
BB1	ビットセット時に分岐
BCND	条件分岐
BR	無条件分岐
BSR	サブルーチンへの分岐
JMP	ジャンプ
JSR	サブルーチンへのジャンプ
RTE	例外からの復帰
TB0	ビットクリア時にトラップ
TB1	ビットセット時にトラップ
TBND	境界チェック時のトラップ
TCND	条件トラップ

ある命令は割り込み受け付け不可となり，アトミック操作を実現できる．88000にはXMEM命令がある．これはコンペアアンドスワップ操作を実現する．またSPARCには2種類のセマフォ命令がある．ロードストア無符号バイト命令は，不可分にメモリをリードしてそこにオール1をライトする．SWAP命令はオール1の代わりに特定のレジスタの値をライトする．

これらを比較すると，i860のロック/アンロック機構がセマフォの実現に適しているようにみえるが，実際にセマフォを実現するとなると複数の命令が必要であり，三つの間で大差はない．

▶ 乗除算

これらのMPUの中では，88000のみが乗除算命令をサポートする．i860には浮動小数点命令の乗算があり，これで代用することができるが，除算はない．SPARCには乗除算のためのステップ命令(部分積などを計算する)がある．i860とSPARCは，ますます乗除算が重要になる当時のアプリケーション状況においては不利な立場にあった．初期のRISCの多くに乗除算命令がなかったことは，半ば常識のようになっている．

しかし，現在では乗除算命令をサポートしないアーキテクチャはまずない．たとえばSPARCは，バージョン8で整数乗除算命令が定義された．

このように乗除算命令の有無が比較対象になるということ自体が，初期のRISCの特徴をよく表している．

▶ 分岐

これらのMPUにはいずれも遅延分岐の概念があり，遅延スロットを利用すると分岐のペナルティの60～70%を削減できる．また，遅延スロットを無効化することも可能で，これはコードサイズの減少に役立つ．また，ハードウェアによる分岐予測をサポートする．

▶ アドレッシングモード

これらのMPUで共通なオペランドのアドレッシングモードは，「ベース＋オフセット」，「ベース＋インデックス」であり，常にゼロを値とするゼロレジスタをもっている．これらを組み合わせると，次の五つのアドレッシングモードを実現できる．

- レジスタ：Rx
- レジスタ間接：(Rx)
- インデックス付きレジスタ間接：(Rx，Ry)
- オフセット付きレジスタ間接：offset(Rx)
- 即値

表12　SPARCの命令セット

算術・論理・シフト命令	
ADD(ADDcc)	加算(と条件コードの変更)
ADDX(ADDXcc)	キャリー付き加算(と条件コードの変更)
SUB(SUBcc)	減算(と条件コードの変更)
SUBX(SUBXcc)	キャリー付き減算(と条件コードの変更)
TADDcc(TADDccTV)	下位の2ビットをタグとみなして加算
TSUBcc(TSUBccTV)	下位の2ビットをタグとみなして減算
MULScc	乗算と条件コードの変更
AND(ANDcc)	AND(と条件コードの変更)
ANDN(ANDNcc)	NAND(と条件コードの変更)
OR(ORcc)	OR(と条件コードの変更)
ORN(ORNcc)	NOR(と条件コードの変更)
XOR(XORcc)	排他的OR(と条件コードの変更)
XNOR(XNORcc)	排他的NOR(と条件コードの変更)
SLL	論理左シフト
SRL	論理右シフト
SRA	算術右シフト
SETHI	rレジスタの上位22ビットをセット
SAVE	呼び出し側レジスタウィンドウの退避
RESTORE	呼び出し側レジスタウィンドウの回復
特殊レジスタ操作命令	
RDY	Yレジスタのリード
RDPSR	PSRレジスタのリード
RDWIM	WIMレジスタのリード
RDTBR	TBRレジスタのリード
WRY	Yレジスタのリード
WRPSR	PSRレジスタのライト
WRWIM	WIMレジスタのライト
WRTBR	TBRレジスタのライト
UNIMP	未定義命令
IFLUSH	命令キャッシュの無効化

ロード／ストア命令	
LDSB(LDSBA)	符号付きバイトロード
LDUB(LDUBA)	符号なしバイトロード
LDSH(LDSHA)	符号付きハーフワードロード
LDUH(LDUHA)	符号なしハーフワードロード
LD(LDA)	ワードロード
LDD(LDDA)	ダブルワードロード
LDFSR	FSRレジスタへのロード
LDCSR	コプロセッサ状態レジスタへのロード
LDF	浮動小数点レジスタへのロード
LDDF	浮動小数点レジスタへのダブルワードロード
LDC	コプロセッサレジスタへのロード
LDDC	コプロセッサレジスタへのダブルワードロード
STSB(STSBA)	バイトストア
STSH(STSHA)	ハーフワードストア
ST(STA)	ワードロード
STD(STDA)	ダブルワードストア
STFSR	FSRレジスタからストア
STCSR	コプロセッサ状態レジスタからストア
STF	浮動小数点レジスタからストア
STDF	浮動小数点レジスタからダブルワードストア
STC	コプロセッサレジスタからストア
STDC	コプロセッサレジスタからダブルワードストア
STDFQ	FQレジスタからストア
STDCQ	コプロセッサキューレジスタからストア
LDSTUB(LDSTUBA)	アトミックなロードとストア
SWAP	レジスタのメモリとのスワップ
分岐命令	
Bicc	整数条件コードによる分岐
Fbfcc	浮動小数点条件コードによる分岐
Cbcc	コプロセッサ条件コードによる分岐
CALL	手続きの呼び出し
JMPL	現在のアドレスをレジスタに退避してジャンプ
RETT	トラップからの復帰
Ticc	整数条件コードによるトラップ
浮動小数点・コプロセッサ命令	
Fpop	浮動小数点命令群
Cpop	コプロセッサ命令群

これらはCISCでもっとも頻繁に出現するアドレッシングモードでもある．

88000では，インデックスをデータサイズでスケーリングすることができる．しかし，そのような使用法は人工知能言語や科学技術計算では有用であるが，通常は使用されない．i860ではレジスタファイルのリードポート数を節約するためにインデックスアドレッシングがない．しかし，CISCマシンでもインデックスアドレッシングの出現頻度は低いので問題ない．

インデックスアドレッシングは，行列計算を効率的に行える．3次元グラフィック用途にはあったほうが望ましい．

▶ レジスタ

これらのMPUはCISCよりも多くのレジスタを提供するが，実際に何本使用できるかはアーキテクチャによって異なる．この意味では88000のレジスタセットは立場が弱い．整数と浮動小数点に共通な32ビットレジスタが32本あるだけである．i860とSPARCは整数と浮動小数点用にそれぞれ32本の32ビットレジスタを提供する．

レジスタの本数が多いi860とSPARCのほうが実際に88000よりよい性能を達成することがわかってい

る．SPARCはこれに加えてレジスタウィンドウもサポートする．

6 現在のRISC

RISCにも20年以上の歴史がある．その中で今も現役として使われているアーキテクチャは，ARM，MIPS，SPARC，PowerPC，PA-RISC，Alphaくらいであろうか．組み込み向けのMPUも実際にはRISCアーキテクチャを採用しており，その中で比較的有名なものは，SHとV850であろうか．

しかし，それらの命令セットをすべて説明することはあえてしない．どれもバークレーとスタンフォードのRISCを基礎とした発展形にすぎないからである．ここでは，これまでまだ詳しく説明していないPowerPC，PA-RISC，Alphaのアーキテクチャに関して簡単に説明しておこう．

● PowerPC

PowerPCは発表当時，RISCの中でも豊富な命令を備えた「Rich RISC」と呼ばれ，（今ではありふれているが）積和命令やレジスタ値に依存した分岐命令，OS専用命令が注目を浴びた．後々高性能化の妨げになるので遅延分岐は採用しないといったのは有名である．

しかもPowerPCのアーキテクチャは最初から完成されており，これまでその命令セットには大きな変更はない．PowerやPowerPCの進化は，命令セットをいかに高速化するかというマイクロアーキテクチャの実装方式の進化である．

最初のPowerPCであるPowerPC601は，次のような高速化技術を採用している．

- スーパースカラ
- 命令プリフェッチキュー
- アウトオブオーダー命令発行
- レジスタリネーミング
- ロード/ストアバッファ

これらは大型計算機の技術をいち早く採り入れたものといえる．

● PA-RISC

PA-RISC（Precision Architecture RISC）とはHP社のEWSであるHP9000シリーズのアーキテクチャであり，EWSの分野ではかなりの実績をもつ．それでいて，ビット操作命令，ビットフィールド命令，独自機能をサポートするSFU（Special Function Unit）を有し，組み込み制御分野にも適している．

PA-RISCの命令長は32ビット固定長で，140種の命令を提供する．その内訳は，メモリ参照命令，分岐命令，算術論理演算命令，システム制御命令，コプロセッサ命令である．命令の特徴は複合機能を有する分岐命令で，加算と条件分岐，比較と条件分岐，転送と条件分岐の機能を1命令で実現し，1クロックで実行する．そのほかにシフトと加算を1クロックで実行する．さらに，演算と分岐命令には次の命令を無効化する機能がある．これにより，遅延スロットを最適化したコードサイズの圧縮やループプログラムの高速化を行うことができる．

たとえば，

```
      ADD    r2,r3,r4
      Bcc    Next
      命令1
  Next:
      命令2
```

という命令列は，

```
  ADD,= r2,r3,r4
  命令1
  命令2
```

と1命令少ない命令列で実現できる．また分岐命令は，前方（forward）分岐の場合は分岐が成立すると遅延スロットが無効化され，後方（backward）分岐は分岐が不成立のときに遅延スロットが無効化される．この機能を利用すると，

```
  LOOP:
      ADD   r1,r2,r3
      ADD   r3,r4,r5
      CMP,<> r6,r5,LOOP
      NOP
```

というループを表す命令列は，

```
  LOOP:
      ADD   r1,r2,r3
      ADD   r3,r4,r5
      CMP,<>,n r5,r5,LOOP+4
      ADD   r1,r2,r3
```

という命令列に置き換えられる．つまり，遅延スロットに分岐先の命令をもってくることができる．これにより，4クロックかかっていた1回のループを3クロックに縮小することができる．これはMIPSのBranch Likelyと同じ考え方である．

HP自体はEWS用のMPUをIntelのIA-64に移行す

ることを表明しているので，PA-RISCが幻のアーキテクチャのまま終わってしまう可能性は大きい．とはいえ，IA-64の開発遅れに危機感をもっているのはHPも同様らしい．2000年に発表されたロードマップでは，2001年に800MHz動作のPA-8700，2002年に1GHz動作のPA-8800，そして将来的には1.2GHz動作のPA-8900の開発を行うことが明示されている．PA-8900以降は完全にIA-64に移行する予定であるが，これはItanium，Mckinleyに続く第3世代であるDeerfieldやMadisonのあとということになる．

HPはPA-RISCからIA-64への移行は非常に簡単だと言っている．なぜなら，IA-64の命令セットのほとんどはPA-RISCのものであり，バイナリレベルの互換性があるためという．これに加えて，データの互換性(エンディアンが同じということ？)もあることが特筆すべきこととして挙げている．

● Alpha

Alphaは当初から64ビットアーキテクチャを提供し，64ビットのロード/ストア命令を基本として命令セットが構築されている．8ビット/16ビットのロード/ストア命令はなく，必要な場合は専用命令でバイトの挿入/抽出を行う(2代目の21164ではこの制限は

Column 現在におけるCISC命令セットの意義

● CISCとRISCのプログラムサイズ

RISC命令セットは，MPUの性能を追及してきた成果である．現在ではほとんどのMPUがRISCになっている．それでは，CISC的な命令セットに意義がなくなったのかというと，そうでもない．性能よりもプログラムサイズのほうが重要視されるROMベースの組み込み制御分野では，いまだにCISC的な命令セットが重宝されている．このような分野では限られた容量のROMにどれだけ多くの機能(＝命令)を詰め込むことができるか否かによって価値が決まる．つまり，プログラムサイズ至上主義である．

現在では，組み込み制御分野もCなどの高級言語を使ってプログラムが記述される．そしてそこで現われる命令機能はかなり定型的である．たとえば，スタックポインタを基準としたメモリ(変数)参照，スタックフレームの生成と破棄，そのスタックフレームへのレジスタの一括した退避と回復，データ型に応じた符号拡張やゼロ拡張などである．これらを複数の基本命令で，パイプライン的に，高速に実行するのがRISCであり，1命令で比較的低速に実行するのがCISCである．明らかにCISCのほうがプログラムサイズは小さい．また，RISCの分岐遅延スロットも，場合によっては命令数を増加させる傾向にあるので，プログラムサイズの観点からはなくてもかまわない．

● ARMのThumb命令セット

ARMの命令セットは，このようなRISCとCISCの命令セットの中間点をうまくおさえているところに圧倒的な人気の秘密があるのかもしれない．とくにARMの命令長を16ビット化したThumb命令セットは，CISC化の傾向が強い．ARM社は，Thumbコードでは40％の性能低下だが，70％のプログラムサイズに圧縮できるとしている．この一見ネガティブな説明がまかり通っているということは，約半分の性能になってもプログラムサイズが重要になる場面があることの証明であろう．

● MIPS16命令セット

ARMと同じように組み込み制御分野に注目しているMIPSも，MIPS16命令セットを定義している．そして，さらにコードサイズを縮小するためにMIPS16e命令セットを定義している．これはMIPS16のスーパーセットで，符号拡張/ゼロ拡張命令，遅延スロットのないジャンプ命令，レジスタの一括退避/回復命令，MIPS32命令の直接実行機能を追加している．ますますCISC色が強くなっている．MIPSライセンシである東芝はMIPS16eを拡張したMIPS16e+を発表し，さらなるコードサイズの削減を目指している．そのおもな拡張機能は，単一ビット操作，ビットフィールド命令，積和命令，飽和命令である．

● ARMのThumb-2

2003年6月，ARM社はThumb命令セットの改良版であるThumb-2を発表した．これは，従来16ビット長のみだったThumb命令セットに32ビット長の命令を混合したものである．ARM本来の32ビット命令とThumbの16ビット命令をモード切り替えする従来方式と異なり，それぞれのビット長の命令の混在を可能とする新しい命令アーキテクチャらしい．ところが，従来の開発ツールの使用が可能で，命令コード自体は従来の32ビット命令やThumbと互換性があると説明されている．これにより，16ビット命令のみの場合より25％の性能向上になるという．プログラムサイズは32ビット命令の74％になる．

ARM社によれば，性能が25％上がった分，動作周波数を下げられるので，低消費電力化が実現できるとしている．ほとんど詭弁(?!)のような説明ではある．しかし16ビット長と32ビット長の命令を混在させることで，Thumb-2はMIPS16により近くなったといえる．

なくなった）．命令長は32ビット固定で，140種の命令がある．

Alpha AXPの特徴はPAL（Privileged Architecture Library）コードにある．PALコードとは，割り込み例外の処理と復帰，コンテキストスイッチ，メモリ管理，エラー処理など，従来はMPUのハードウェアで処理していた機能をMPUのハードウェアを直接操作するサブルーチンで実現する．OSやハードウェア構成の違いごとにPALコードを用意することで，基本となるアーキテクチャが異なるシステムにも共通にAlpha AXPアーキテクチャのMPUを搭載できるといわれている．PALコードは，MPUの実装別に定義されるPAL命令と通常命令で構成される．たとえば，1992年に発表された最初のAlpha21064は，次の3種5命令のPAL命令をサポートする．

- 内部レジスタのリード/ライト命令
- MMUを介さないロード/ストア命令
- PALコードからの復帰命令

PALコードによってアプリケーションプログラムの実行とOSの実行が分離されているため，ハードウェアはアプリケーションプログラムの命令セットを高速実行できるように最適化されている．

7 SIMD命令/暗号化処理命令

● マルチメディア対応命令

SIMD（Single Instruction Multiple Data）とは，一つの命令で複数のデータを処理することを意味する，演算方式を表す言葉である．各プロセッサメーカーは命令セットに独自性を出すために，マルチメディア対応や特定分野対応の命令を追加するのに躍起である．

Intelはi386アーキテクチャに**MMX**（MultiMedia eXtention）テクノロジーという命令セットを追加した．さらにPentium IIIからは**SSE**（Streaming SIMD Extension）を，Pentium4では**SSE2**という命令群を追加している．AMDも同様に，**3DNow!**という命令群を追加している．

そしてMIPSでは，**MDMX**（MIPS Digital Media eXtension：マッドマックスと発音する）というマルチメディア系の命令セットを追加し，そのサブセットがR5432で実装された．また，単精度浮動小数点を並列実行するための**MIPS-3D**という命令セットも発表され，R20000で実装された．

PlayStation2のEmotionEngineのベクトルユニットに実装されているマルチメディア命令群も忘れてはいけないだろう．これは東芝のTX79コアに継承されている．

ARMも2000年のMicroprocessor ForumでSIMD命令の追加（v6アーキテクチャ）を表明した．

SPARCは，64ビットアーキテクチャのバージョン9でマルチメディア系の**VIS**（Visual Instruction Set）を追加した．それはUltraSPARCで実現されている．

整数系ではなく浮動小数点系の強化をしたものには，PowerPCの**AltiVec**もある．

SH-4は最初から浮動小数点のSIMD命令を命令セットとして提供している．SH-5では整数系のSIMD命令も採用されるようだ．2003年6月のEmbedded Processor Forumでは，SH-5のSIMD命令の紹介が行われた．8ウェイのSIMD命令は，MPEG-4のエンコード時などに威力を発揮するという．このように，SIMD命令の採用は花盛りである．

ここでは基本をおさえるという意味で，MMXについて学んでおこう．あらためて見直すと，MMXが提供する機能は，ほかのプロセッサが採用するマルチメディア命令の機能とほとんど同じであることがわかる．そして最後に，ARMとMIPSのSIMD命令に関して少し言及する．

● MMXテクノロジ

MMXの基本的な考え方は，8ビットまたは16ビットの要素を一つの比較的小さなデータにパックして並列に処理することである（**表13**）．具体的には次のような機能を有する．

▶ パックされたデータ形式

MMXでは新しいデータ形式を定義する．マルチメディアアプリケーションで扱うデータの多くは，8ビットまたは16ビットとサイズが小さい．またマルチメディア処理は，多くの隣接するデータ要素を同時に扱うことが多い．MMXでは，これら二つの特色をSIMD処理で実現する．いくつのデータ要素を並行処理すればよいかはアプリケーションの特性に依存するので，1種類には決定できない．ただ，Intelのプロセッサは64ビットのデータパスをもっているので，MMXのデータ型も64ビットと決められた．具体的には，**図1**に示すように4種のデータ型がある．

▶ 条件付き実行

条件によって操作を切り分ける場合，分岐命令を使用することが考えられる．しかし，分岐予測を誤る場合の損失を考慮すると実行速度は遅い．さらに，従来

表13　MMXの命令セット

オペコード	オプション	実行クロック	記　述
PADD[B/W/D] PSUB[B/W/D]	ラップアラウンド 飽和	1	パック化データの加減算を並列に実行
PCMPEQ[B/W/D] PCMPGT[B/W/D]	一致 より大	1	パック化データの比較を並列に行い，マスクを生成
PMULLW PMULHW	結果が下位 結果が上位	レイテンシ 3 リピート 1	パック化ワードデータの乗算を並列に行い，結果の上位または下位を選択
PMADDWD	16ビットから 32ビットへの変換	レイテンシ 3 リピート 1	パック化ワードデータの乗算を並列に行い，隣接する32ビットの結果を加算
PSRA[W/D] PSLL[W/D/Q] PSRL[W/D/Q]	シフト量がレジスタか即値か	1	パック化データの算術論理シフトを並列に実行
PUNPCKL[BW/WD/DQ] PUNPCKH[BW/WD/DQ]		1	パック化データをインタリーブしながら混合
PACKSS[WB/DW]	常に飽和	1	パック化データを並列に生成
PLOGICALS		1	ビット単位の論理演算
MOV[D/Q]		1	転送
EMMS	実装依存		FPレジスタのタグを空にする

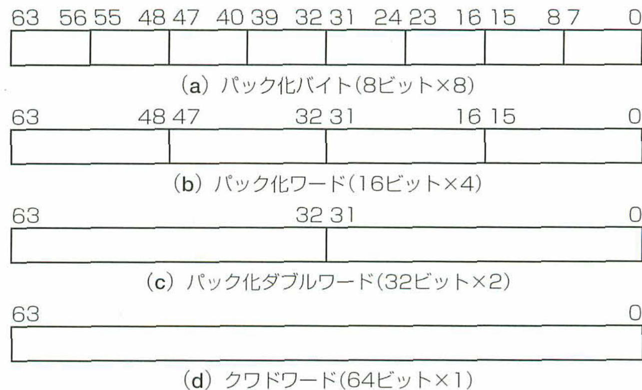


図1　MMXのデータ型

の手法を適用しようとすると，パックされたデータ型をスカラ型(組になってない形式)に変換する必要がある．

これを解決するのが条件転送(条件に応じて転送するデータを切り分ける)である．しかし，このためには三つの独立したオペランド(ソース，デスティネーション，各データ要素に対する条件の組)が必要なので，2オペランドを基本とするインテルアーキテクチャではつごうが悪い．条件転送(条件代入)にはいろいろな実装方式があるが，MMXではマスク付きの代入を採用する．しかし，これには三つのオペランド(ソース，デスティネーション，マスク)が必要である．

そこで，MMXではマスク生成と代入を2段階に分離した．専用の比較命令が各オペランドに対応するビットマスクを生成する．たとえば比較処理の場合は，八つのバイトがパックされたオペランドに対して，八つの8ビットのマスクを生成する．そのマスクを論理演算と併用することで条件転送を実現できる．**図2**に，四つのワード(32ビット)要素に対する比較操作を示す．

▶ 飽和演算

マルチメディアで典型的に使用されるオペランドサイズは小さい．たとえば，RGBαという色の各要素は8ビットで表現される．8ビットあれば256階調の色が表示できる．これは人が認識できる解像度を超えているが，問題点もある．8ビットでは多くのピクセルの色を蓄積していくと，8ビットで表現できる上限を超えてしまうのだ．デフォルトの設定では，二つの数値の加算結果が上限値を超える場合はラップアラウンドする．つまり，結果が8ビットで表現できなくても，下位8ビットをそのまま値とする．しかし，メディアアプリケーションでは，そのようなオーバフローに対する防御策が必要となる．具体的にはラップアラウンドせず，最大値に留まることが望まれることもある．

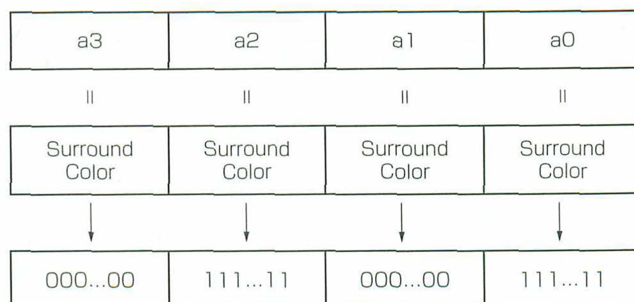


図2
ワード型に対するパックトイコール

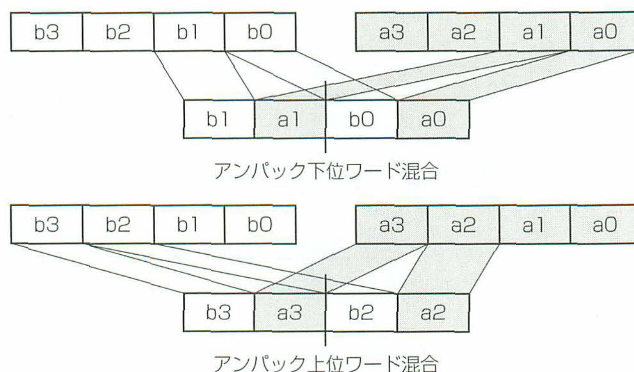


図3　MMXのアンパック命令

▶ **固定小数点演算**

メディアアプリケーションでは，フィルタ処理などにおける重み付け係数を扱うため，小数点処理が必要となる．これに対応するため，浮動小数点のSIMD処理を提供することも考えられる．しかし，浮動小数点処理はハードウェアの負担が大きいため，実際のメディアアプリケーションでは10〜12ビットの精度で，動的には4〜6ビットの範囲が表現できれば十分である．

このような状況を鑑み，MMXでは固定小数点演算をサポートすることとした．固定小数点演算は加減算に関しては整数演算と同一視できるが，乗除算に関しては整数演算を適当にスケーリング(右シフト)しなければならない．これらの機能をサポートしているわけである．

しかし，3D分野でのジオメトリ変換など，単精度浮動小数点の演算精度が必要なアプリケーションもあるのも確かである．これらは，後年，SSEやSSE2で実現される．

▶ **パックされたデータ型のデータ要素の並び替え**

アプリケーションによっては，パックされたデータ内での要素の並び替えや，二つのパックされたデータのマージが必要である．一般的には二つのパックされたデータをオペランドとし，デスティネーションに任意の順序で各バイトを混合することを許す．

しかし，これでは実装が複雑になる．MMXでは，アンパック命令により，パックされたデータの要素の並び替えと結合を行うことができる．この命令の動作を図3に示す．この命令を使用すれば，ピクセルのパック型の形式をプレーン型の形式に変換できる．

▶ **積和演算**

マルチメディアや通信アプリケーションにおいて，もっとも頻繁に出現するのが積和演算である．これは行列の乗算やフィルタ操作の基本操作として使用されるためだ．積和演算を用いる行列-ベクタ操作の例を図4に示す．

● ARMのSIMD命令

ARMは2000年秋のMicroprocessor Forumで，既存の命令セットに追加されるSIMD命令の概要を発表した．IntelのMMXと同じように互換性の維持を第一に考え，パイプラインの実行に影響を与えない18の命令を定義した．このため，命令機能は，加算，減算，選択，乗算，飽和に関するものに限られている．そのほかの命令機能はARMのコプロセッサインターフェースを通じて使用できる．これはStrongARM(またはXScale)のSIMD命令の実装と同じである．ただし，互換性はない．

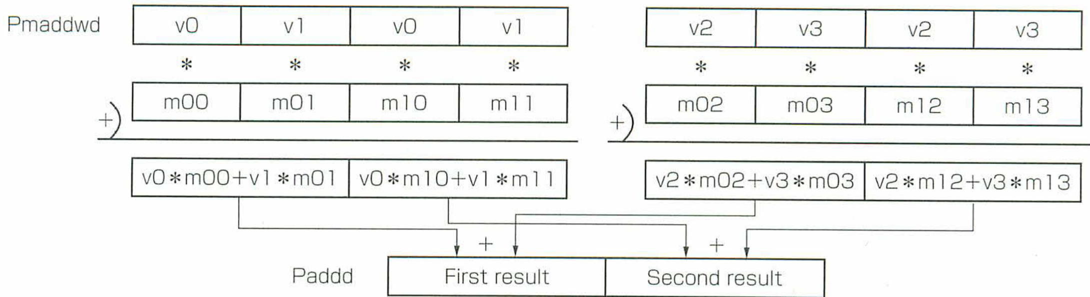


図4 行列-ベクタ演算

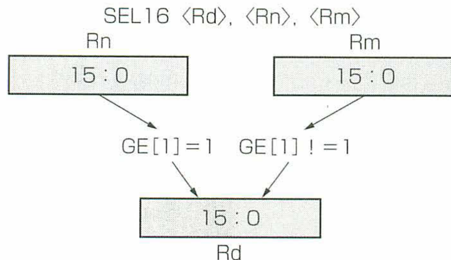


図5 新しい条件フラグを使う選択演算

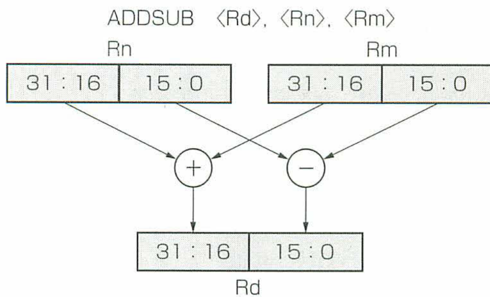


図7 SUBADD命令

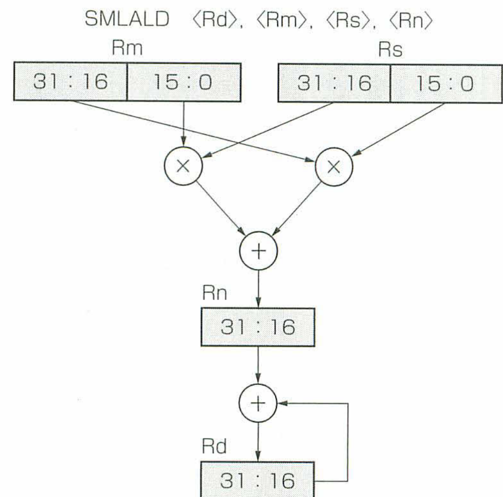


図6 積和演算の例

ARMのSIMD命令の特徴的なところは，従来のハードウェア資源を利用して命令を拡張したことである．特殊レジスタの追加も，ベクタ処理をサポートするALUも追加していない．ただし，SIMD命令のために新しい状態フラグを定義する．それがGE［3：0］で，プログラムステータスレジスタのCPSR［19：16］にマップされている．**図5**はこの条件フラグを使う選択(SEL)演算の動作例である．

SIMD命令では16×16ビットの積和演算をサポートするのが流行であるが，ARMもその例にもれない．**図6**はARMの積和演算を示す．SMLA{X}D命令は二つのレジスタの上位16ビットと下位16ビットどうしをたすきがけに乗算し，その積を加算する．そしてその結果の上位または下位16ビットをもう一つのレジスタの上位または下位16ビットと加算する(交換処理)．これは，フィルタ処理や複素数の積の計算に有効である．

また，有用なSIMD命令として加算-減算(ADDSUB)，減算-加算(SUBADD)命令がある．これは，**図7**に示すように，16ビットのデータどうしで行われる．この操作は，FFTやDCTの変換処理に使用できる．

● MIPS-3D ASE

MIPS-3D ASE(Application Specific Extension)は，MIPS社が提唱している3次元グラフィックアプリケーションを高速に処理するための拡張命令セットである．これは，3次元ジオメトリ処理のために，従来の命令セットに新たな13命令を追加したものになっている．従来の単精度，倍精度浮動小数点のデータ型のほかに，ペアドシングル(Paired Single)，ペアドワード(Paired Word)というデータ型が新設された．ペアドシングルとは，一つの64ビット浮動小数点レジスタに二つの単精度浮動小数点データを格納するものである．ペアドワードとは，一つの64ビット浮動小数点レジスタに二つの単精度固定小数点データを格

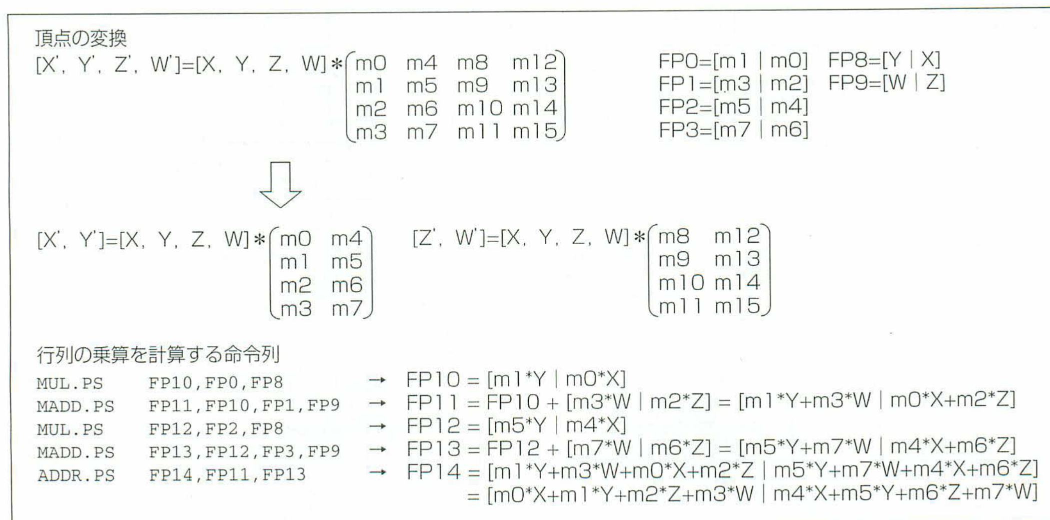


図8　ADDR命令の使用例

```
頂点が指定領域にあるかテスト
|X| ≦ |W|
|Y| ≦ |W|
|Z| ≦ |W|
```

```
PUU.PS      [W|W]
NEG.PS      [−W|−W]
C.NGE.PS    !(Y ≧ −W)?  !(X ≧ −W)?   →条件コード CC0,CC1
C.NGE.S     !(Z ≧ −W)?               →条件コード CC2
C.LE.PS     (Y ≦ +W)?  (X ≦ +W)?     →条件コード CC3,CC4
C.LE.S      (Z ≦ +W)?                →条件コード CC5
BC1F    CC0, 範囲外      条件 (CC0) 不成立なら範囲外
BC1F    CC1, 範囲外      条件 (CC1) 不成立なら範囲外
BC1F    CC2, 範囲外      条件 (CC2) 不成立なら範囲外
BC1F    CC3, 範囲外      条件 (CC3) 不成立なら範囲外
BC1F    CC4, 範囲外      条件 (CC4) 不成立なら範囲外
BC1F    CC5, 範囲外      条件 (CC5) 不成立なら範囲外
```

(a) 従来方式での計算

```
CABS.LE.PS    (|Y| ≦ |W|)?  (|X| ≦ |W|)?
              →条件コードCC0,CC1
CABS.LE.PS    (|W| ≦ |W|)?  (|Z| ≦ |W|)?
              →条件コードCC2,CC3  (CC2は真)
BC1ANY4F  CC0,CC1,CC2,CC3,範囲外
              →  条件が一つでも偽なら範囲外
```

(b) 新しい命令での計算式

図9　CABSとBC1ANYnxの使用例

納するものである．これらは2ウェイのSIMD方式での処理(要は並列実行)を可能にする．

MIPS社によれば，MIPS-3Dを用いると，もっとも内側の処理ループのコードサイズを30%削減できるので，1秒間に処理できるポリゴン数が45%増加するそうだ．頂点の座標変換での行列の乗算を高速化するために，ペアドシングルリダクション加算命令(ADDR)が定義された(**図8**)．リダクションとは，行列とベクトル間の乗算の部分的な乗算処理を指すらしい．画像のクリッピングは，ペアドシングル絶対値比較命令(CABS)と多重条件コード分岐命令(BC1ANYnx)によって簡略化できる(**図9**)．

透視変換には逆数命令(RECIP1，RECIP2)が使用できる．また光源処理には逆数平方根命令(RSQRT1，RSQRT2)が使用できる．これらの逆数演算はMIPS64アーキテクチャにあるが，より高速に実行できる．

そして，ペアドシングルとペアドワード間でデータ変換を高速に行う命令(CVT.PS.PW，CVT.PW.PS)も

表14 MIPS-3Dで拡張された命令

ニーモニック	処理
ADDR	浮動小数点リダクション加算．組どうしの加算
MULR	浮動小数点リダクション乗算．組どうしの乗算
RECIP1	逆数．高速近似値．精度的には劣る
RECIP2	逆数．第2ステップ．精度を上げる処理
RSQRT1	平方根の逆数．高速近似値．精度的には劣る
RSQRT2	平方根の逆数．第2ステップ．精度を上げる処理
CVT.PS.PW	ペアドワードからペアドシングルへの型変換
CVT.PW.PS	ペアドシングルからペアドワードへの型変換
CABS	浮動小数点絶対値比較
BC1ANY2F	二つの条件コードのどれかが偽なら分岐
BC1ANY2T	二つの条件コードのどれかが真なら分岐
BC1ANY4F	四つの条件コードのどれかが偽なら分岐
BC1ANY4T	四つの条件コードのどれかが真なら分岐

ある．

追加された13命令の概要を**表14**に示す．

● **暗号処理命令**

暗号といえば，従来はICカードやスマートカードの機密保持に使用するものであった．しかし，ネットワークが普及するにつれて，ネットワークを介したデータ転送の暗号処理機能が重要になってきた．従来は外付けのコプロセッサで対応していたが，より高速な処理を達成するため，暗号処理の基本機能をMPUの命令として提供することが考えられている．

本来，暗号は自動化(ハードウェア化)が困難なように構成されるので，命令セットでサポートするのは無謀であるともいえる．しかし，純粋にソフトウェアで記述するよりも20～100倍の性能向上が期待できるので，MPUの特色を出すためには回路規模を犠牲にしても採用する意義がある．

2002年6月，Sun MicrosystemsはUltra SPARC Vに，暗号処理機能やネットワークのプロトコルスタックの処理機能の搭載を検討していることを明らかにした．IBMのPower5でも同様の命令の導入が予定されている．

また，暗号処理機能をサポートするのは暗号エンジンの高速化だけでは不十分ということで，暗号を解く鍵や機器に固有のID番号などの情報を格納する特殊メモリ空間や，乱数生成をハードウェア機能として提供するMPUも登場し始めた．これらの機能は従来ならMPUの外部ロジックで実現されていたが，セキュリティを強化するためには，その機能をMPU内部に取り込む必要がある．

2003年1月にはTransmetaが従来からのTM5800に，暗号化エンジンをはじめとするセキュリティ機能を組み込むことを表明した．具体的には，DESやDES-X，3DESのアクセラレータと，保護されたメモリ領域を内蔵するらしい．

2003年4月に出荷されたVIA Technologiesの新しいC3(Nehemiah)では**PadLock**と呼ぶセキュリティ機構(ノイズを利用したハードウェアによる乱数発生器と暗号化エンジン)を搭載している．

2003年5月には，ARM社がARM11以降のMPUでは**TrustZone**と呼ぶセキュリティ機能を内蔵することを表明している．これは，Monitorモードという新しい動作モードを定義し，この動作モードでのみ保護したアドレス空間へのアクセスを可能とする機能である．

8 MPUの今後

● **RISCの終焉？**

現代においてもっとも普及しているMPUはx86(CISC)であり，RISCは終焉(しゅうえん)を迎えつつあるという見方がある．System InsiderというWebサイトで，Massa POP Izumida氏(元x86アーキテクトという)が，「頭脳放談」という連載の第27回「RISCの敗因，CISCの勝因」〔参考文献1)〕で次のように述べている．

① RISC登場の背景は，単純な命令を高速に動かせば高性能が得られるという考えに基づく．これはハードウェアを単純化することから始まった
② しかし，さらなる高速化要求のために大規模な回路(スーパースカラや多重レベルキャッシュなど)が必要になり，回路が複雑になった
③ 互換性を維持しながら新機能を追加することも複雑化の一因である

④ 結果，開発に多くのリソース(人的，物的)が必要になった
⑤ CISC(x86)は，PCの普及が巨大な市場を形成し，リソースを費やしただけの見返り(利益)が期待できる
⑥ RISCはその基盤であるEWS市場の成長が小さく，注ぎ込んだリソースに見合う見返りがない
⑦ CISCにアーキテクチャ上の問題があったとしても，注ぎ込むリソース差が性能差(とくにクロック)や価格差に表れ，RISCよりも有利な状況にある
⑧ 単純に技術要素のトレードオフを論じるならRISCのほうが合理的であるが，現実世界ではきわめて偏りのある「ハンデ戦」で競争が行われている
⑨ EWSの存在意義は既存設計資産が使えるということのみで，CAEツールベンダがコスト/パフォーマンスの高いPC + Linux対応の製品を投入してきている現在では，EWSの将来性は危うい
⑩ コンピューティング分野では，性能が上がらないRISCは青息吐息の状態
⑪ RISCが全滅かというと否で，組み込み制御の世界には「高速化=リソース競争」とは違う競争原理が働いているため，RISCの生き残る道はある

かなり的を射た意見であるとは思う．しかし，Massa POP Izumida氏とは異なり，筆者はコンピュータをRISCとCISCに分類して議論するのは無意味と考える．

実際，現在CISCと呼ばれるコンピュータも内部はRISCであり，たいていは命令デコード部と実行部が切り離される構成を採っているので，命令セットの違いはデコードが終了するまでになくなってしまう．結局，高速化を追及すると行き着く先は同じものになると思われる．CISCが勝ってRISCが負けたのではなく，両者が融合したと考えるほうが自然だと思う．また，Massa POP Izumida氏は「性能=動作周波数」と考えているふしがあり，やはり「x86な人」なのだと思う．

● 最近のプロセッサ事情

最近，SoCではおもしろい(？)現象が起きている．プロセッサの性能はバス転送能力のみで決定され，CPUコア単体の性能は全体の性能に寄与しないというのである．極論すればCPUコアは何でもよく，周辺機能として内蔵されるメモリコントローラやPCIコントローラの性能で，全体のシステム性能が決定するというものである．

これは，ある意味では真実であろう．しかし，種々のサービスを提供するためには，プロセッサ性能は必要である．実システムでMPU(CPUコア)の性能が支配的であるか否かは，そのシステムが稼働するまではわからない．あるいは，周辺ユニットの動作周波数の都合から，性能はともかくCPUコアの動作周波数が決定されることもある．

つまり，CPUコアの動作周波数はバスの周波数と同じ，または整数倍であることが望ましい．この時点の要旨は，CPUコアの性能ではなく，いかに周辺デバイスとの共同で動作できるかである.性能ではなく，動作周波数が要求される．このような状況ではプロセッサのマイクロアーキテクチャは意味をなさなくなり，動作周波数のみが議論の的となる．これは，性能向上のための動作周波数の向上とは別の次元の話である(キャッシュのヒット率は重要ではあるが)．

現在では，CPUコア単体の性能がいくら良くても意味をなさない時代になっているのかもしれない．

まとめ

過去から現在に至るMPUの命令セットアーキテクチャを見てきた．アーキテクチャには，CISCとかRISCという区別はあるものの，それらが提供する命令セットにはたいして違いがないことがおわかりになったと思う．MPUができることは昔も今も変わらない．ただ，SIMD命令による並列処理が新たな潮流といえるかもしれない．この傾向は，他社との差別化のために，今後ますます強くなっていくだろう．

また，一つのアーキテクチャを維持するのはたいへんなことである．システムの環境整備にはばくだいな金額がかかる．いきおい，ハードウェアは従来との互換性を維持しようとし，ソフトウェア(とくにOS)のサポートのないアーキテクチャは滅びていく．Alphaが滅びIA-64が生き延びていく傾向は，それをよく表している．

参考文献

1) Massa POP Izumida，「頭脳放談 第27回 RISCの敗因，CISCの勝因」(`http://www.atmarkit.co.jp/fsys/zunouhoudan/027zunou/end_of_risc.html`)
2) H. Kaneko et al., "Realizing the V80 and Its System Support Functions", *IEEE Micro*, April 1990, pp.56-69
3) R.S.Piepho et al., "A Comparison of RISC Archtectures", *IEEE Micro*, August 1989, pp.51-62

Epilogue

研究段階から実用化へ，そして現在残っているのは……

RISCプロセッサ興亡史

● RISCはCISCに対するアンチテーゼ

RISCはCISCに対するアンチテーゼとして開発された．その発想は，CISCではむだな命令が多く，それが動作周波数を上げる妨げになっているというものだ．洗練された簡単な命令セットにすれば，パイプライン処理を効率的に動作させることができ，動作周波数の向上とあいまって最高の性能を達成できる．この観点で，1980年代から，大学や企業でさかんにRISCの研究が行われるようになった．

ある時期まで，CISCは思いつきでアーキテクチャを拡張していった感があるが，RISCは最初から定量的に性能評価をしながら論理的にアーキテクチャを決定していった．

● CISC……いや，x86の一人勝ち？

RISCは，本来は動作周波数を上げるための技術であるが，現時点ではx86系MPUのほうが動作周波数が高い．結局は，プロセス技術の開発に莫ばくだいな投資をできる企業だけが生き残ることができるのだ．投資回収という観点から見れば，需要がしれているサーバやEWS向けのRISCの利益が少ないのは明らかで，ばくだいな数が出荷されているPC用のMPUには太刀打ちできない．金持ちはより金持ちに，貧乏はより貧乏に，もしくは論理派が現場からの叩き上げに負けたようなものであろうか．

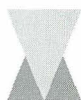
しかし，x86がRISCの技術を採り入れながら性能(特にIPC)の向上を図ってきたのは事実であり，その上でプロセス技術によってさらなる高速化がなされたと考えることができる．その意味で，x86の進化にあって，RISCは高速化のためのアイデアを提供してきたとも言える．Intelと並ぶ優秀なプロセス技術を有するIBMは，自社のRISCであるPowerPCで一人気を吐いているが，ほかのRISCはサーバ/EWS/PCなどの高性能が要求される分野からは早晩消滅していくかもしれない．

● RISCの進むべき道

RISCの進むべき道としては，性能よりも超低消費電力が要求される組み込み制御の分野であろう．CISCは性能を追求するあまり，消費電力には無頓着になっている．最近こそIntelなども電力削減のスローガンを掲げているが，ARMやSHやMIPSなど，早い時期から低電力を売りにしてきたRISCには追いつけない．GHzを超える動作周波数を維持しようとする限り，永久に追いつくことは不可能であろう．そこにRISCの生きる道があると思われる．

RISCには，x86のようにPCを中心とした共通の応用分野があるわけではなく，各コンピュータメーカーが自社のサーバやEWSを高速化できればいいという発想で開発が続けられたので，そのアーキテクチャは各社バラバラである．したがって，その解説はオムニバス形式にならざるをえない．

ARMやSHやMIPSは，組み込み制御分野を自分の色に塗り変えようと必死の努力をしているが，x86のように世界統一できる日が来るのだろうか．



RISCプロセッサの興亡

以降ではRISC系MPUの興亡史を簡単にまとめてエピローグに代える．

● IBM801

1975年，IBMは高級言語のプログラムを用いて既存のものよりも非常にコスト/パフォーマンスのよいシステムを提供するため，ハードウェア，OS，コンパイラを含めたコンピュータシステムの設計に着手した．その設計思想はほとんどの命令の処理は1クロックで終了するべきというものだった．これは，コンパイラが生成する命令コードは基本命令(ロード，ストア，分岐，比較，加算)ばかりであり，それらを高速化すれば全体の性能が向上するという根拠に基づく．その研究の結果，生まれたのがIBM801である．しかし，なぜかIBMは1982年までその存在を公表しなかったという．

IBM801の扱うデータ型はキャラクタ(バイト)，ハーフ

ワード，ワードの3種類で，アドレッシングモードはベース+インデックス+オフセットという単純なものだった．命令長は1ワード固定で，3オペランド演算を行った．また，当時は汎用レジスタの本数は16本が主流だったが，IBM801は32本のレジスタを備えていたのが特徴的である．IBM801はECLプロセスで製造され，決してVLSIと呼べる集積度ではなかったが，命令キャッシュなどRISCの特徴を備えており，後年のカリフォルニア大学バークレー校やスタンフォード大学のRISC研究に大きな影響を与えた．

その割りには，IBM自身はRISCの有効性に気付いていたとは言い難く，IBMがRISCに本格的に手を染めるのは，1992年のPowerPCになってからである．

● バークレーRISC I/RISC II

カリフォルニア大学バークレー校のRISCの研究は，高級言語コンパイラが複雑な命令を有効に使えないことに着目したことから始まった．プログラム実行時の命令の出現頻度，アドレッシングモード，変数の使われ方などの統計を採り，新しい命令セット設計の指針とした．この研究結果は，1980年に同大学のPattersonとDitzelによって初めてのRISCに関する論文『The Case for the Reduced Instruction Set Computer』として発表された．この論文はシングルチップコンピュータに最適なアーキテクチャはRISCであると主張し，バークレー校の大学院生がその理論を実践したのがRISC IとRISC IIである．これらは当時のCISCよりも単純で，設計の労力も少なかったが，CISCに匹敵する性能を発揮していた．かくして，バークレーRISCは後のARMやSPARCアーキテクチャの基礎となるのである．また，RISCという言葉はバークレー校によって初めて使用された．

ちなみにDitzelは，Crusoeを開発したTransmetaの創業者としても有名である．RISCを提案した彼がCrusoeではVLIWを選択したところが興味深い．

● スタンフォードMIPS

バークレーRISCと同時期，スタンフォード大学でもHennessy教授を中心にRISCの研究が行われていた．それは以前のRISCとは毛色が違い，ハードウェアをできるだけ単純にして，レジスタの依存関係等の煩雑な処理はリオーガナイザと呼ばれる再構成ソフトウェアで解決しようと試みた．パイプラインを効率的に動作させるためにハードウェアによるインターロックを許さず，ソフトウェアで命令の順序を並び替えて正常動作を保証する．

このためスタンフォード大学のRISCは『Microprocessor without Interlocked Pipeline Stages(パイプラインステージがインターロックしないマイクロプロセッサ)』の頭文字を採ってMIPSと呼ばれた．もちろん，コンピュータの性能を示すMIPS(Million Instructions Per Second)との掛け言葉である．

MIPSではハードウェアを簡単にするために，メモリアクセスはワードアクセスのみとし，バイト単位の操作が必要な場合は専用命令を使ってレジスタ上で処理する．また，汎用レジスタは16本だった．

最初のMIPSプロセッサは実用的といえるものではない．しかし，スタンフォード大学の研究者たちは，その研究を推し進め，2Kバイトの内蔵命令キャッシュと256Kバイトの外付け混合キャッシュインターフェース，32本の汎用レジスタ，乗除算用の特殊レジスタ，ゼロレジスタ，5段パイプラインを特徴とするMIPS-Xというプロセッサを設計した．これは後のR2000とほぼ同じ構成であるが，分岐の遅延スロットが2命令である点が決定的に異なる．

● R2000以降のMIPS

スタンフォードMIPSの研究成果を受けて，1984年にベンチャー企業のMIPS Computer Systems(現MIPS Technologies Inc.)が生まれ，R2000を始めとする非常に高速なプロセッサファミリを世に送り出すことになる．これは，Hennessy教授の研究成果を誰も信じてくれなかったため，RISCで高性能を達成できることを実証するために自らMIPS社を創設したものである．

1991年，MIPS社は世界初の64ビットMPUであるR4000を発表する．CPU，FPU，L1キャッシュ，L2キャッシュインターフェース，マルチプロセッサ機能を1チップに内蔵した画期的なものだった．

MIPSのMPUはそれまで主にSGI(Silicon Graphics Inc.)のGWSに採用されていたが，SGIは1992年にMIPS社を買収して超高性能なMPUの開発を行わせることになる．それがR10000を始めとするハイエンドプロセッサシリーズである．R10000の開発と並行して，MIPS社はローエンドのR4200/R4300も開発している．R4300はNintendo64に採用されることを目的として開発されたMPUである．

この頃になると，元MIPS社の開発メンバがスピンアウトしてQED(現PMC-Sierra)，SandCraftといった新しい会社を設立し，MIPS社とは別個にMIPSアーキテクチャのMPUを開発するようになった．

そして1998年，SGIが自社のGWSのMPUとして将来的にIntelのIA-64プロセッサ(Itanium)を採用することを決定すると，MIPS社はライセンス管理とIPコアの販売を目的としてSGIから分社化される．分社化されたMIPS社は，それまでのMIPSアーキテクチャを，32ビットのMIPS32，64ビットのMIPS64として再整理した．そしてそのほかに

も，特定分野向けにMIPSアーキテクチャの拡張を積極的に行うようになった．たとえば，組み込み制御用に16ビット長の命令でコード効率を上げるMIPS16，3次元グラフィックス用のMIPS-3D，スマートカードやJava用のSmart MIPS，ネットワークアプリケーション拡張用などを提唱している．

現在では，SGI，MIPS社以外でも，独自にMIPSアーキテクチャのプロセッサを製造するようになっている．代表的なところでは，PMC-Sierra，IDT，NEC(V_Rシリーズ)，東芝(TXシリーズ)，Alchemy，SiByte(現在はBroadcomに買収された)がある．特にAlchemyとSiByteはDECでAlphaチップやStrongARMチップを開発していた技術者がスピンアウトして設立した会社である．いわば，ほかのアーキテクチャからの乗り換え組である．

2002年2月，Alchemyはx86互換メーカーの老舗であるAMDに買収された．AlchemyはAu1000/Au1500というMPUで，400MHzで500mW，500MHzで900mWと，高動作周波数にもかかわらず低消費電力を達成していることで定評がある．IntelがStrongARMやXScaleでインターネットアクセス系の組み込み分野に進出しようとしているのに対抗する形である．IPコアの分野において，MIPSはARMよりもやや不利な状況にあったが，この買収劇でMIPS陣営が活気づくかもしれない．さらに2002年4月29日，AMDはMIPS64のライセンスを取得した．AlchemyはMIPS32のライセンスしかもってなかったが，これでAMDは思う存分MIPSチップを製造できる．

さらに2003年8月6日，AMDは組み込み向けx86プロセッサであるGeodeをNational Semiconductorから買収した．Au1000シリーズの運命はどうなるのかと思ったが，MIPSプロセッサは低消費電力が重要な携帯機器分野で生き残るという．Geodeの消費電力は7W程度，Au1000シリーズの消費電力は0.5W程度である．

● MIPSも周波数競争に参入

2002年10月のMicroprocessor Forumでは，Broadcomが1GHzのMIPS64コア(SB1)を1チップに4個内蔵するBCM1400を発表した．消費電力は，1GHz動作時に1チップ全体で25Wとかなり小さい．同フォーラムで東芝は800 MHz動作のTX99(Amethyst)をCPUコアとするSoCであるTMPR9961を発表した．CPUコアはMIPSでありながら，内部バスがARM系のAMBAバス(AHB)であることが興味深い．また同時期，NECはV_R5500(コードネームSapphire)というCPUコアで800MHz動作を達成したと発表し，2003年には1GHz版の開発を終了すると発表した．PowerPCやx86系のMPUに対して周波数の向上が遅れていると酷評されることが多いMIPSアーキテクチャであるが，着実に動作周波数を上げてきている．

もっとも，MIPSアーキテクチャプロセッサで1GHz動作という発表は以前からあり，PMC-Sierraは2001年のMicroprocessor Forumで1GHz動作のRM9000x2を発表した．BoradComも直後にSB1を2個内蔵するBCM1250を発表した．しかし，いずれも1GHz動作時の消費電力は非常に大きく，1GHzで使用している顧客は少ないと聞く．ちなみに，消費電力のカタログ値は，RM9000x2が5W，BCM1250が6W(3W × 2)と小さめである．しかし，PMC-Sierraは低電力化のためにシングルコアのRM9000x1を開発した．BroadcomもPCM1250のシングルコア版であるBCM1125を発表している．PMC-SierraにしろBroadcomにしろ1GHz動作を表明しているが，その消費電力の高さからか使い方は難しいようである．

2002年4月にはIntrinsity社が2GHz動作(！)のMIPS32プロセッサであるFastMIPSとFastMATHを発表しているが，これもどうだろうか．消費電力は10～20Wというが，2003年に消費電力を下げた1GHz動作のFastMATH-LPを発表しているところが実に怪しい．

2003年10月31日には，PMC-Sierraは，改めて(？)，1 GHz動作のRM9000x1GLシリーズを発表した．ギガビットEtherMAC，最大500MHz対応のDDR-SDRAMコントローラ，Hyper Transportを内蔵して，消費電力は7W未満という．この消費電力ではx86プロセッサと競合する応用分野しか残されてないと思うが，そうなれば(x86よりはIPCが高いMIPS系の特徴を考慮したとしても)1GHzではまだまだ性能不足かと思われる．

● MIPSの将来は…

現在，MIPS64系のIPコアは5Kのみとなり，20K(Ruby)と25Kf(Amethyst)が新しいロードマップからは消え失せている．これは，MIPS社がMIPS32アーキテクチャに注力しようという表れであろう．新しいプロセッサの24Kという名称は明らかに25K(Amethyst)の下位チップであることを強調するものである．スーパースカラではなくなったが，シングルパイプで動作周波数を向上させていくことで20Kや25Kの置き換えをねらったものであろう．東芝はMIPS社と共同でTX99(25Kf，Amethyst)の開発を行っていたが，これでは梯子を外された格好である．

これをもって，MIPS社はMIPSアーキテクチャを管理・維持していく求心力を失ったと見る向きもある．2003年5月にはSandCraft社も倒産(のちにRaza Microelectoronics社が買収)しており，64ビットアーキテクチャ(MIPS64)を維持していくためには，NEC，東芝，PMC-Sierra(QED)，

AMD(Alchemy), Broadcom(SiByte)の頑張りが期待される.

さて, 2003年10月にはSGIが自社サーバをItanium2で統一していく方針を決定しており, R18000の開発は中断したという噂が流れている. R16000の高速版(1GHz?)を開発してMIPSプロセッサの開発は集約する方向と聞く. MIPSの将来はますます危うしといったところか.

ただ, 標準品としてのMIPSプロセッサの販売は難しくなったが, SoCのCPUコアとしてのMIPSは健在でありつづけると考える. これで, ライバルのARMの売り方にますます似てきたといえる. とはいえ, ARMが1社でアーキテクチャを支えているのに対して, MIPSは上述のライセンシー各社が共同で支えているという感じである.

● ARM

ARMの歴史は, 英BBC教育チャネルによる教育用コンピュータプロジェクトに端を発する. このプロジェクトがAcorn Computerに発展した. 1983年, Acorn社は英国で成功を収めていた6502を搭載したBBCコンピュータ(一般にはBeepとして知られている)の次機種用のMPUを探していた. しかし市場に出ていたMPUで満足のいくものがなかったため, Acorn社はMPUを自社開発することにした. しかし, MPU設計の経験が浅かったAcorn社は少ない設計労力で良い設計をする必要があった. そこに現われたのがバークレーのRISC Iの論文であり, それを参考に, 約18ヶ月を費やしてMPU設計がされた. これがAcorn RISC Machine, つまりARMである.

ARMプロセッサは, 当時PDA Newtonを計画していたAppleの目にとまり, 同機のCPUとして採用されることになった. これが契機となり, 1990年にARMは, Apple, Acorn, VLSI Technologyの出資によって設立されたAdvanced RISC Machines社に移管された. そして, その3年後にNewtonが発売された.

ARMはARM7でヒットを飛ばし, ARM8, ARM9とアーキテクチャを開発していく.

後年, ARM社はDEC(Digital Equipment Corporation)と共同でStrongARMを開発した. これは, 高速動作を誇るDECのAlphaプロセッサの技術を応用して高い性能を引き出そうとするものである. ARMとしてはキャッシュ構成に初めてハーバードアーキテクチャを採用した.

一方, StrongARM部門はIntel社に買収された. 正確には, 1997年にDEC社がIntelをAlphaの知的所有権侵害として提訴した際の和解条件の一環として, DEC社が半導体工場をIntelに売却したとき, 工場と一緒にARMの製造権利も付いてきたのだ. Intelにすれば PC用MPU以外の組み込み制御分野への事業展開を図るための手駒をおまけ同然(?)に入手したわけである. この状況に面白くないのはARM社である. ARM社自体はARM9, ARM10と独自の製品展開を行っていくが, StrongARMに関しては口を閉ざして何も語らなかった.

ところで, ARMアーキテクチャの版数はARMプロセッサの種類と密接に関係している. 基本的に, ARM6と前期のARM7がv3, 後期のARM7, ARM8, ARM9がv4, ARM10がv5である. なお, ARM1, ARM2, ARM3がArcon社によって作られたMPUであり, ARM6以降がARM社によって作られたMPUである. ARM4, ARM5はAcornからARMへの変更過程で欠番となった.

現在最新のアーキテクチャであるv6の特徴を一言でいうと, SIMD命令の追加によるマルチメディア拡張である. 注目すべき点は, キャッシュの構成が仮想アドレスキャッシュから物理アドレスキャッシュに変わったことであろう. 物理メモリの有効活用が目的というが, 互換性の問題は大丈夫なのだろうか. MIPSと同様, このところのARMアーキテクチャもゴチャゴチャしたものになってきたと思うのは筆者だけか.

ARM社はARMの製造ライセンスを多くの半導体メーカーに与えているが, 基本的に, ARMアーキテクチャの改造権はない. ARM社の提供する設計情報をそのまま使用しなければならない. これが, ICE機能をも含め, 確固たる互換性を実現している. MIPSが売りにしている独自命令拡張機能も, 応用分野を限定するものであり, 互換性を損なうとして, 導入するつもりはないようだ.

しかし, StrongARMを製造するIntelにはARM社の強制力はない. Intelは独自にStrongARMを改造して, より高速, より低消費電力なStrongARM2(後にXScaleとして発表された)を開発してしまった. しかしXScale登場当初は, バグが直らず散々だったようだ.

そして2001年7月30日, ARM社は次世代のプロセッサアーキテクチャv6を, IntelとTexas Instruments(TI)にライセンスすると発表した. ARMとIntelは不仲だったはずでは? しかし, Palmが次期OS用のデバイスとしてARMアーキテクチャを採用するにあたり, Intel, TI, Motorolaと提携することを表明したのを契機に和解したのかもしれない(真相は不明). XScaleの(バグや性能が低いという)評判を受けて, ARM社がXScaleを脅威に感じなくなったせいというのは穿ち過ぎか.

別の情報筋によれば, Intelは元々ARMアーキテクチャをカスタマイズする権利を有していたという. 今回契約を拡大し, v6だけでなく, ARM7, ARM9, ARM10のコア

にアクセスする権利を得たという．ARMとIntelの不仲説は幻想だったのかもしれない．実際，いつの間にかXScaleの命令セットはARM v5TE互換ということになっている．これはXScaleも純粋なARMの系譜であることを強調するものである．現在，StrongARMとXScaleがARMのWebサイトに載っていることを思うと感慨深い．

2002年4月29日，ARMはEmbedded Processor ForumでARM11コアを発表した．これは，v6アーキテクチャに基づく最初の製品となり，2002年第4四半期にライセンスを開始するという．XScaleをARMの一員と認めつつも，本家ARMの意地にかけて，より高性能のMPUをぶつけてきたというところか．

ところで，ARMはMIPSとは異なり，熾烈な周波数競争に参入する気配はない．2002年の末にSumsung社が1.2 GHz動作のARM10を発表したが，これは特異な例であろう．本家のARMは組み込み制御分野では，動作周波数と消費電力のバランスが重要と考えているようだ．動作周波数は製造プロセスのトレンド(0.13 μmプロセスなら550 MHz程度)を維持しつつ，いかに低消費電力化するかに注力しているように見える．

ARM社によると，2004年にはARM11の次のMPU(ARM12？)を製品化するとしている．これは，0.13 μmルールで600MHz以上，90nmルールで800MHz以上の動作周波数を予定しているらしい．

● XScale

XScaleは，ARMと同様に，Intelが独自設計したARM互換のCPUコアの名称である．ただ，当初は(今も？)マイクロアーキテクチャの名称だった．周辺機能を交換することでいろいろな種類の製品に展開できる．ARMが，SoCのCPUコアとなることを目的としていて，カスタムチップ形態で供給されるのとは反対に，XScaleは特定の周辺機能を内蔵した標準品として供給される．XScaleの名称の由来は，明らかにExtended Scaleであり，動作クロックが数MHzから1GHzまでと広い範囲に使えるCPUコアを提供するという意気込みが表れている．

XScaleの開発の契機となったのはWindowsCEマシン用に新MPUが必要になったことである．最初，WindowsCEマシンに搭載されていたのはStrongARM(SA1100)というMPUで，190MHzという高速動作の割りに330mWという低消費電力が特徴だった．ただ，消費電力に重点を置くあまり，性能は二の次となっている感があった．ベンチマークテストによる性能は惨たんたるものだった．性能の出ない(公式な？)理由はバス転送能力が低いということで，バス速度を向上させたSA1500という製品が計画されていたが消息不明である．

その後，1999年後半からリリースされたWindowsCE3.0(Rapier)に搭載されたのはSA1110というMPUである．これはSA1100の高速版(206MHz動作)である．バスサイクルの周波数が103MHzと高速なので，それなりの性能を発揮した．従来機と比較して「信じられないほどの高性能」と評した人もいた．しかし，対抗のMIPSやSHと比較すると，相対的な性能は大したことなかった．

WindowsCEのStrongARMの悪評を打ち消すため，1999年ごろからIntelが本気になった．DECによるStrongARMの設計を放棄し，自社開発に踏み切ったのだ．その性能たるや，600MHzで動作し，CPUコアの消費電力が0.1mW(1チップでは450mW)というもので，Intelでなかったら戯言としか受け取られない高性能である．この数値を競合他社はかなり脅威に感じているはずだ(ただ，実際の性能は600MHz動作で約700MIPSと控えめではある)．有言実行できるか否か，その答えは近いうちに出るはずだった．しかし，次期StrongARM(SA2)の開発には3年以上の歳月を要してしまった．

2000年8月23日，Intel Developer Forumにて次期SA2の発表があった．が，蓋を空けてみると話が大きくなっていた．SA2は，第三世代(3G)の携帯電話機用に開発したXScaleというマイクロアーキテクチャを実装し，最高1GHzの周波数で動作するという．しかも，そのときの消費電力はわずか1.5Wというから驚きである．性能は約1300 MIPSとか．

しかし，ターゲット市場の本命は，あくまでも携帯電話であることを匂わせていた．この場合，50MHzで動作させることになり，そのときの消費電力は10mWという．これは従来のMPUなら待機時並の低電力である．この時点では，SA2がWindowsCEに使われる目はなくなったと見られていた．

ところが，WindowsCEには採用されないという風評に反し，2002年2月12日，IntelはXScaleをコアとするPXA250とPXA210というプロセッサを日本で先行発表した．PXA250が200/300/400MHzの3品種，PXA210が133/200MHzの2品種となっている．しかし，実際にWindowsCE(PocketPC2002)に採用されてたのは400MHz品ばかりだった．WindowsCEではOSが重過ぎて，200 MHzや300MHz動作ではまともな性能が出ないらしい．

しかし，そのPXA250も思ったほどの性能ではなかった．パイプラインの段数が増加したこともあるが，バス速度が100MHzと低いことが最大の原因といわれた．ベンチマークテストによると，PXA250の400MHz品とSA1110の206

MHz品の体感性能にほとんど違いがなく，ユーザーを失望させた．

PXA250に関してはキャッシュのライトバック機能にバグがあり，ライトスルーで操作させていたために性能が低下しているとの悪評があったが，2003年3月25日，そのバグを修正したPXA255が正式発表された(WindowCE用には，その少し前から流通していた)．PXA255ではバス速度も200MHzに向上され，それなりの性能になったため，ユーザーの信頼を取り戻すことに成功した．

PXA255の発表と同時に，Flashメモリを1チップ内に集積する，いわゆるスタックまたはSiP(System In Package)という構成を採るPXA263も発表された．これにより，デバイス内の部品数を減らすことができ，メーカーは製品のサイズ縮小や新機能搭載が可能になる．

この頃から，XScaleのロードマップが明確になってきた．XScaleプロセッサをWindowsCEやPalmOSといった携帯機器だけでなく，ワイヤレススマートディスプレイや携帯メディアプレーヤーにまで拡大することを模索している．

携帯電話向けにはPAX800F(コードネームManitoba)がある．312MHzのXScaleプロセッサ，4MバイトのFlashメモリ，104MHzのDSPをSiPに封入する．

また，ネットワーク分野にも，EthernetMACやUSBを内蔵したIXPシリーズというXScaleプロセッサを搭載したチップが投入されている．とくに，IXP422とIXP425は，暗号の補助機能が内蔵されており，その低価格戦略のゆえか，広範囲に採用されている．動作クロックには266/400/533MHz版があり，その高性能さもアピールしている．

さらに，次世代携帯機器向けに，PXA255の後継機種のBulverdeと呼ばれるXScaleプロセッサに取り組んでいることも表明している．ワイヤレスMMXとMobile Scalable Linkという新機能が搭載され，動作クロックも500MHz～600MHzになるという．性能的にはPXA255の1.5倍以上といわれている．

● i860とi960

i860は，1986年に開発が着手され，1989年に発表されたIntel初のRISCプロセッサである．1988年の半導体技術を考えると，100万トランジスタを1チップに集積できるとし，x86とは異なる新しいアーキテクチャを提供する目的で開発された．特徴として，二つの32ビット命令(整数と浮動小数点)を64ビット命令とみなして，1度に命令キャッシュから取り込んで同時に実行するデュアル命令モードを備える．これをもって，1チップVLIWの第1弾という栄誉を受けている．

i860はベクトル浮動小数点演算器やMMXの原型ともいえるグラフィック命令を内蔵していた．結果，i386＋80387の性能を上回る高性能を実現することができ，従来のスーパーコンピュータやミニコンが提供していた科学技術計算や各種のシミュレーションをより小型で安価なシステムで実現できた．

ただ，i860では，命令を正確に実行する責任(ハザード回避)はハードウェアよりもむしろコンパイラに任せていた．結果としてi860は成功しなかったが，すべてソフト任せというハードウェアの自由度の低さが一因だったのでなかろうか．結局，コンパイラがうまく開発できなかったのだろう．

i960は，1988年に発表されたIntelの組み込み制御向けのRISCプロセッサである．i960KAを基本とし，FPUを内蔵したi960KB，マルチタスク，MMU，フォールトトレラント機能をサポートするミリタリーグレード(軍用)のi960MCがあった．1989年にはi960KAをスーパースカラ構造にしたi960CAも開発されている．

i860/i960とIntelが立て続けにRISCプロセッサを発表したため，世の中の風潮はRISCになるのではと思われたが，それほど流行らなかった．ほどなく，当のIntelが80486を発表してRISC対抗を打ち出したところに方針の矛盾を感じる．

i860/i960は現在でも，シリーズ名は若干変更されたが，DSPの代用品やRAID用のプロセッサとして生き残っている．

● 88000

EWS用のMPUがRISCに傾き始めるとMotorolaもその風潮に迎合した．1988年から88000の出荷を開始した．当時の副社長は「混乱した32ビットプロセッサの競争で生き残ったのはIntelとMotorolaだ．RISCも同様だ．」と自信満々のコメントを出していた．

88000とはCPUであるMC88100とキャッシュとMMUを内蔵するMC88200というチップの総称である．コードユニット，データユニット，整数ユニット，FPU(加減乗除と変換用の二つ)の計五つのユニットが各自パイプラインで並行動作するという意味でスーパースカラのはしりである．1988年に発表された．MC88100は比較結果を反映させる条件コードレジスタをもっていない．比較命令は，ほかの演算命令と同じく，3オペランド命令で，比較結果をデスティネーションレジスタに格納する．条件分岐命令はこのレジスタの値に基づいて分岐する．この構成により，比較命令と条件分岐命令間の命令を自由にスケジューリング(入れ替え)できる．条件コードを使用しないこの方式は，

MIPSを初めとする多くのRISCで採用されている．

しかし，88000自体はすでに開発を終了し，Motorolaの主流はPowerPCに移行している．88000の売れ行きは，最初は好調だったが，徐々に翳りを見せていった．88000の失敗の原因はいくつかあるが，最大の原因はIntelのi486対抗の68040と並行して開発をしたため，リソース不足となり次機種の開発が大幅に遅れてしまったことである．実際，88000と68040そのものも開発遅れとなった．特に68040の開発遅れはEWSメーカーからの採用が相次いでキャンセルとなり，RISCに移行するきっかけを与えてしまった．

● SPARC

SPARCとは『Scalable Processor ARChitecture（拡張性のあるアーキテクチャ）』を意味する．それ自体は命令セット，アドレッシング，MMUのソフトウェアのみの規格であり，実装は半導体メーカーに任されている．

現在のSPARCは，1992年にSun Microsystemsがバークレー RISCの研究成果を基に，自社のEWSであるSUN4用のMPUのアーキテクチャとして提唱し開発したものである．その仕様はオープンアーキテクチャとしてSPARCインターナショナル社によって管理されている．そのメンバーはSunのほかに，富士通，TI，LSI Logic，Rossなどで，各社が独自にSPARCチップの開発を行っている．

SPARCにはいくつかのバージョンがあり，最新バージョンは9である．バージョン9は64ビットアーキテクチャであるが，（少し前の）典型的なSPARCチップは32ビットアーキテクチャのバージョン7または8に基づいている．

2001年ごろから，SUN OS（Solaris）の64ビット化が急速に進行中で，MPUもバージョン9を実装するUltraSPARC Ⅲの750MHz品を搭載したマシンがSUNの稼ぎ頭になってきた（なお，UltraSPARCシリーズの製造はTIが担当している）．またサーバーやEWS分野だけでなく，組み込み向け分野では，低消費電力版SPARCとしてSPARCLite（富士通）などもシェアをもっている．

SunはIA-64のItaniumをライバルと考えている節がある．しかしIA-64同様，実際の製品出荷は発表したロードマップからは遅れぎみだ．Sunでは，現在，別々の開発チームがUltraSPARC ⅣとUltraSPARC Ⅴの開発に従事している．UltraSPARC Ⅴの動作周波数は1.5GHzを匂わせているが，1GHz動作のUltraSPARC Ⅳも出荷されてない現状では，まだ海のものとも山のものともわからない．なお，2003年のMicroprocessor Forumで，SunはUltraSPARC Ⅳは2004年6月までに1.2GHzで登場すると発表している．UltraSPARC Ⅳは，UltraSPARC Ⅲを2個内蔵するチップで，UltraSPARC Ⅲと端子互換になる．

なお，富士通は自社のサーバに搭載するためにSunとは別個に，SPARC V9（バージョン9）を実装するSPARC64シリーズを開発している．2003年に出荷している最新チップは1.35GHz動作のSPARC64 Ⅴである．Ⅴというのは第5世代のSPARCチップを示している．

富士通のロードマップによると，2004年後半には2CPUコアで2.4GHz動作のSPARC64 Ⅵ，2006年には4CPUコアで5GHz以上の動作のSPARC64 Ⅶを出す予定である．これは，Sunのロードマップとほぼ一致している．ただ，技術的には，Sunよりも富士通のほうがやや進んでいると思われる．

2003年10月22日付けの日経新聞に富士通とSunがハイエンドUNIXサーバの開発に関して協力し，将来的には事業統合も視野に入れているというニュースが一面に掲載された．両者とも否定のコメントを出しているが，富士通もSunも赤字続きであり，事業統合は当然の流れであろう．

2003年10月のMicroprocessor Forumでも，Sunと富士通はなぜ別々にSPARCチップを開発しているのかという質問があったそうだ．それに対するSunの回答は，「IntelとAMDに合併しろというようなもの」だとはぐらかしたという．

2004年2月11日，SunはUltraSPARC Ⅴへのつなぎとして，従来はチップの外にあった，L2キャッシュを同一シリコン上に集積したUltraSPARC Ⅳ＋（コードネームPanther）を発表した．UltraSPARC Ⅳ＋の性能はUltraSPARC Ⅳの2倍になる見込みだ．

● スループットコンピューティングとNiagara

2003年2月，SunはUltraSPARCのロードマップを再度強調した．これは，SunがIntelに敗退するというアナリストの予想を牽制する意味がある．発表の中心はUltraSPARC Ⅳだったが，コードネームがGeminiと呼ばれる低電力版UltraSPARC Ⅳを搭載したローエンドサーバを2004年に出荷するとした．

さらに，Afara Websystemsの買収で獲得した「スループットコンピューティング技術」を採用するNiagara（コードネーム）が90nmプロセスで2005年に登場するということにも言及した．これはUltraSPARC Ⅱ相当の4スレッド同時実行するマルチスレッドプロセッサを8個搭載する32ウェイプロセッサである．アナリストは，UltraSPARC ⅥでUltraSPARC路線とスループットコンピューティングが統合され，UltraSPARC ⅣやUltraSPARC ⅤはSunの歴史の中で脚注（特に言及する価値のないプロセッサということか）に過ぎなくなると見ている．

2003年8月のHotChipsシンポジウムでは，Geminiこそがスループットコンピューティングの第一弾であることが公表された．Gemini(双子)の名称の通り，2個のUltra SPARC IIと1MバイトのL2キャッシュを1チップに集積したチップである．チップはUltraSPARC IIIiと端子互換になっており，UltraSPARC IIIiのプラットホームに差し込んで既に動作しているという話である．130nmプロセスで製造され，動作周波数は0.9～1.2GHzであるという．しかし，これは当初言っていたUltraSPARC IVそのものではないのだろうか．そうでないとしても，UltraSPARC IVとの違いは，CPUコアがUltraSPARC IIかUltraSPARC IIIかということになる．それが低電力版たるゆえんなのだろうか．ともかく，予定より早くUltraSPARCとスループットコンピューティングの統合が始まっていると考えられる．NiagaraはUltraSPARC Vとして登場してくるのだろうか．

Sunは2月11日にAfara Websystemsで買収した技術とJavaアクセラレータのMAJCの技術を融合したRockというプロセッサを開発すると発表した．ただし，性能が1.2 GHzのUltraSPARC IIIの30倍になること以外，詳細は不明である．これがNiagaraだろうか．

● Am29000

AMD(Advanced Micro Devices)は，今やIntelの最大のライバルであるx86互換機メーカーである．しかし，昔はIntelの8080A，8086，80386やZilogのZ8000のセカンドソースを製造する会社というイメージがあり，オリジナルなMPUには注力してなかった．

唯一，1978年に発表されたビットスライス型のAm2901ファミリがあった程度である．これらはコンピュータや専用制御回路を作成するためのLSI群だった．たとえば，Am 2901は4ビットALUやレジスタを含むLSIであり，8ビットプロセッサでは2個，16ビットプロセッサでは4個を組み合わせて使用する．

そのAMDが，1987年に突如，オリジナルな1チップRISC型MPUを発表した．それが，バークレーRISCの流れを受け継ぐAm29000である．これは，同じバークレーRISCの流れを継ぐSPARCと同時期である．CISCで言えば，Intelの80386やMotorolaの68030と同世代である．

Am29000の特徴は，192個という大量の32ビット汎用レジスタである．これが，SPARCと同様のレジスタウィンドウ方式で使用され，64本がグローバルレジスタ，残りがスタックキャッシュとなる(SPARCのグローバルレジスタは8本)．命令体系は3オペランド形式で，分岐遅延スロットをもつ．この時期のRISCと同様に乗除算命令はもっていないが，乗除算処理の1ステップ分を実行する命令が用意されている．これもSPARCとよく似ている．Am 29000は4ステージパイプラインのスーパースカラ構造を採る．また，浮動小数点演算を行うためにはコプロセッサのAm29027が必要である．

Am29000の命令セットの特徴は，この時期のRISCには珍しく，CISCと同様なビット操作命令とビットフィールド命令をもつことだ．そのためか，初期のPostScript対応プリンタで大量に使われている．また，Macintosh II fx発表時には，Apple Display Card 8・24GCに採用され，「本体より高速なCPUによるアクセラレータ付きのカード」として話題になった．Am29000は22MIPS(30MIPSという発表もある)の性能(30MHz動作)であり，Macintosh本体のMPUであるMC68030の12MIPSを遥かに凌いでいたからである．

Am29000は，EWSというよりも組み込み制御向けに使われたので，同時期のRISC系MPUの中では出荷数が多い．しかし組み込み分野で，徐々にMIPSやSPARC系のMPUに押され始めたAm29000は，1995年，突如「終了宣言」が出された．登場も突然だったが，終焉も突然だった．以後，AMDはAm29000の技術を応用してK5を開発し，より利益の見込めるx86互換ビジネスに注力するようになる．

● Transputer

かつて一世を風靡したTransputerは，英国INMOS社が設計した，マルチプロセッサシステムの構築を容易にするMPUである．INMOSは，1978年に英国の国策会社として設立された．設立初期はRAMを開発していたが，1980年に最初のTransputerを開発した．Transputerは，Transister とComputerを併せた造語である．INMOSは1984年に民営化されている．1987年にはSGS-Thomson Microelectronics(現在のSTMicroelectronics)に買収されている．

INMOSは，1978年にオックスフォード大学のCharles Antony Hoare博士によって提唱されたCommunicating Sequential Processes(CSP)理論に基づいて，Occamというコンピュータ言語を開発した．Transputerは，このOccamのプログラムを実行させるために開発され，その後の並列コンピュータの実用化に多大な影響を与えた．

Transputerは単純さを第一として開発された．たとえば，レジスタは3個しかなく(この他に，FPUに3個，命令ポインタ，ワークスペースポインタ，オペランドレジスタの計9本がある)，命令セットも単純である．内部的にはスタックアーキテクチャに基づいたRISCである．

Transputerは最大4チャネルの双方向シリアルリンクをもち，これを他のTransputerのシリアルリンクと上下左右に結合することで，並列的なネットワークシステムを構

築できる．ネットワーク形態には，相互通信，パイプライン，ツリー，2次元格子，ハイパーキューブなどがある．このシリアルリンクとチップ内のメモリはDMAで通信される．

Transputerの最上位版であるT9000は1991年に発表された．T9000は，最高50MHzで動作し，8命令を同時実行するスーパースカラである．一説によると，このT9000の開発がうまくいかなかったため，Transputerのラインが途絶えてしまったという．

Transputer自体は簡単に並列システムが構成されるために，大学などの研究機関で多用されていた．しかし，通常のMPUでマルチプロセッサ構成が常識になっている現在では，その存在を知る人も少ない．

● CRISP/Hobbit

CRISC(Complex RISC)という言葉がある．これはCISC命令セットをRISCアーキテクチャで実行するPentiumなどに与えられた名称である．しかし，それと同様な試みはRISCの創世紀から存在した．時期的には80286から80386，あるいはMC68020の時代である．その中でも特に有名なのが，CRISP/HobbitとCLIPPERである．これらについて簡単に説明する．

AT&T MicroelectronicsのHobbit(ATT92010)は，1975年に始まったベル研究所のC言語マシン計画に基づくMPUである．C言語に最適化されたCRISP(C-machine Reduced Instruction Set Processor)というアーキテクチャを採用する．CRISPはバークレーRISCやCrusoeで有名なDavid R. Ditzelらによって開発された．CRISP自体はCRISPプロセッサとしていくつもの論文が発表されている．

C言語はスタックを多用する言語なので，CRISPはスタックを基準としたメモリとメモリの演算に最適化されている．いわゆるスタックマシンで，ユーザーから見えるレジスタは存在ない．その代わり32エントリのスタックキャッシュを備える．スタックマシンは数式の処理を簡略化でき，C言語を簡単に実装できる．

また，命令プリフェッチ&デコードユニットとRISCライクな実行ユニットがパイプラインで動作し，大抵のメモリ演算を1クロックで処理できる．命令は16ビット可変長でVAX-11/780並みのコード効率を誇る．これにより，16MHzのHobbitの性能はVAXの約13倍，MIPS R2000の約1.2倍の性能を実現した．当時としては非常に高性能の部類に入る．また，当時のMPUが5V動作だったのに対してHobbitは3.3V動作品もあり，低消費電力をも実現していた．AT&TはCMOS 3.3V動作を特に強調し，MIPS/Wattという，現在ではPDAの性能を示すのに多用される単位を初めて使用したとされる．

Hobbit(CRISP)は，分類的にはCISCであるが，アーキテクチャが非常に単純化され，分岐の最適化など非常にエレガントな特徴も有する．これをもって，AT&TはHobbitをRISCと呼んでいた．1993年，ARMを採用するApple社のNewton対抗として，EO社のPersonal Communicatorという電話機一体型ペンコンピュータに採用された．

Hobbitのロードマップとしては，1993年に25MHzのATT92020，1994年に30MHzのATT92030，1995年に40MHzのATT92040が発表されていた(いずれも3.3V動作時)．

HobbitはNECがセカンドソースをするという話があり一躍有名になったが，ATT92020以降が出荷されたという話は聞かない．

● CLIPPER

CLIPPER C100は1986年にFairchild社によって開発された．その中心人物はスーパーコンピュータで有名なCray Research社のHaward Zaxらである．基本的にはUNIXエンジンを目指し，その上でのCAD/CAEを効率的に処理することを目的としている．後年，EWSメーカーのIntergraphが開発を引き継ぎ，1988年に上位機種のC300が開発された．

C100はMotorolaの88000と同様に3チップで構成される．つまり，CPU，命令キャッシュ(命令TLB込み)，データキャッシュ(データTLB込み)からなる．88000やPA-RISCは汎用の32本のレジスタをもっていたが，C100は16本のユーザー(整数)レジスタと8本の浮動小数点レジスタをもつ．命令長は16ビット，32ビット，48ビットの3種がある．基本的にはロード/ストアアーキテクチャのRISCであるが，直接アドレスも指定できる．コード効率はVAX-11/780の約1割増しである．これはCISCのコード効率に近い(非常に良い)．

また，CLIPPERはスーパーバイザモードに移行すると，ユーザーレジスタがすべて16本のスーパーバイザレジスタに切り替わる(ARMのレジスタバンクはFIRQ割り込みでユーザーレジスタの半分がスーパーバイザレジスタに切り替わる)のが特徴的である．

C100の性能はVAX-11/780の約5倍であり，場合によってはVAX8000よりも高速という．C300の性能はC100の2倍である．性能的には頑張っているほうか．

CLIPPERの開発当時のロードマップは，1987年に動作クロックを33MHzから40MHzへ向上し，その後に10MIPSのCLIPPER IIを開発するとなっていた．しかし，1986年10月にFairchild社は富士通に買収された．富士通

は当時TRONチップを開発していたのでCLIPPERの開発はIntergraph社が引き継いだ．Intergraph社はCLIPPER IIともいえるC300を世に送り出したが，その後が続いていない．

CRISP/HobbitにしろCLIPPERにしろ，比較的高性能を実現したが，現在では歴史の中に埋もれている．やはりそのアーキテクチャを支える企業の体力がIntelやMotorola，RISCの四強(SPARC，POWER，MIPS，PA-RISC)に劣っていたのが敗因か．

● 60xからG4までのPowerPC

PowerPCとは，IBMが大型計算機用に設計したPOWER(Performance Optimized With Enhanced RISC)アーキテクチャを1チップで実現するものである．本来はIBMのEWSであるRISC System/6000(RS/6000)のアーキテクチャを基にしている．RS/6000は科学計算を前提として設計されたが，PowerPCはノートPC，組み込みコントローラ，高性能科学計算用およびGWS(Graphics Work Station)，マルチプロセッサ構成のメインフレームなどに対象を拡げた．

RS/6000は7～9の複数チップから構成されたが，PowerPCでは1チップの実装に適するように命令の削除，追加が行われている．特に使用頻度が低く，IPCを上げにくい命令は削除または簡潔化されている．追加命令は整数や浮動小数点の演算系に多い．

PowerPCは発表当時，RISCの中でも豊富な命令を備えた「Rich RISC」と呼ばれ，(今ではありふれているが)積和命令やレジスタ値に依存した分岐命令，OS専用命令が注目を浴びた．後々高性能化の妨げになるので遅延分岐は採用しないといったのは有名である．

1991年7月にApple，IBM，MotorolaがPowerPCに関して提携し，IBMとMotorolaが製造するMPUをIBMやAppleが自社のPCに採用することを決めた．IBMとAppleは，IntelとMicrosoftの寡占状態にあるPC分野において，新しいPCだけでなく情報機器の世界標準を作ろうと画策して，Motorolaを巻き込んだ形で提携を成立させた．Apple，IBM，Motorolaの連合は各社の頭文字を取ってAIM連合(後にはPowerPC連合)と呼ばれるようになった．AIM連合の成立から約1年後，1992年の6月にはテキサス州のオースチンにSomersetという研究所が設立されて本格的な開発が始まった．その名称の由来はアーサー王の宮殿のあった場所であり，AIMの各社を円卓の騎士になぞらえたのではないかと思われる．事実，PowerPC750(コードネームArthur)の発表時がSomersetの最盛期だった．

1992年12月，IBMはすでに最初のPowerPCアーキテクチャのMPUとして，PowerPC601を開発していた．Somersetの最初の仕事はPowerPC601のシュリンク版の開発である．それは1994年に登場した．IBMの公式資料によれば，まず，POWERを1チップ化したRSCというMPUがあり，それを基にPowerPC601が開発されたとある．このあたりの事情は不透明である．その後，IBMとMotorolaは，PowerPC601に引き続いて，第2世代(G2)のPowerPC 603/604/620をを開発することになる．さらに第3世代(G3)のPowerPC740/750を開発することを表明した．

PowerPC604までは32ビットモードのみをサポートする．PowerPC604を基にして64ビットモードをサポートするPowerPC620が1994年に発表された．しかし，製品としては登場していない．これは欠点を補って後述のPower3となりサーバ向けに使用された．

当初，Somersetでの製品開発は非常に順調にいっているように見えた．PowerPC連合は次期PCの標準仕様としてPRep(PowerPC Reference Platform)やCHRP(Common Hardware Reference Platform)という規格を制定して，PowerPCの市場への浸透を図った．1996年にはPC向けにPowerPC603eとPowerPC604e(604の1次キャッシュ倍増版)が発表され，Macintoshの68000系からPowerPCへの完全移行も発表された．そして，1997年に開発コードネームArthurと呼ばれていたPowerPC750(G3)が登場すると，MacintoshもPowerPC603/604からPowerPC750に移行していった．

しかし，PowerPC連合の蜜月時代は長くは続かなかった．AppleがIBMの意向を無視してPowerPCをApple(Macintosh)に特化しすぎたため，IBMはMotorolaとの共同開発から撤退してそれぞれ独自のPowerPC路線を構築することになる．具体的には，性能向上のためのAltiVecの仕様をめぐる意見の対立が原因といわれている．また，MicrosoftがWindowsNTでのPowerPCのサポートを途中で放棄したことも一因であろう．

その後AppleとMotorolaは，1999年8月に，第4世代に相当するPowerPC G4(MPC7400)を発表した．これはPowerPC740/750に，結局はMotorolaが開発した，マルチメディア用SIMD命令であるAltiVecを実装した製品である．G3，G4はMotorolaが開発したPowerPC603をコアとした派生品であり，どれも最大2命令(＋AltiVec)同時発行，最大3命令同時実行のスーパースカラということになっている．

● Power4とPowePC970

IBMにはPowerPCとは別のロードマップにサーバ向けのラインアップとして，Power1，Power2，Power3があ

る．これらはPowerPCを，より性能が出るように，SMP(Symmetric Multi-processor System)対応に改造したものである(Power1はPowerPC601と同一という話もあるが)．IBMの大型計算機であるRS/6000シリーズの開発チームによって独自に開発された．しかし，これらのチップとそれを搭載したサーバは，動作周波数が他社に比べて劣るため，市場でのシェアを徐々に失っていった．

Power4はそのような状況を打破するために開発された．Power4は，命令セット以外は，それ以前のPowerチップとは別物である．高速クロック技術は637MHzを実現したPowerPCの開発チームが担当した．高速バス技術，パッケージ技術も，それぞれIBM内の専門家チームが担当した．システム設計はRS/6000チームが担当した．CPUコアはPower3の技術者とSomersetの残留組が担当した．このように，その時点でのIBMがもてる最高技術を注ぎ込んで開発されたのだ．

Power4の特徴の一つは低消費電力であることである．従来，サーバに使用されるMPUは性能が第一で消費電力は二の次になっていた．しかし，電力削減のためPower4では，チップ内部の使用中でない部分をオフにし，必要な時には即座にオンにできる特殊な回路を採用した．組み込み系MPUでしばしば採用されるマスククロックの考え方だ．IBMは自社の前世代のチップ(450MHzのPower3)，およびほかのUNIX市場のライバル製品との比較で，半分程度しか電力を使わないと主張している．

なお，Power4の予想消費電力はISSCC2001で1.5V駆動時に115W(1.1GHz)と発表されている．各CPUコアは30Wの消費電力だそうだ．当時のPowerPCの最新版であるPowerPC750CXeの消費電力が6.0W(600MHz)であることを考えると，さすがにサーバ向けという感じである(電力使いまくり)．ちなみに，競合他社のCPUの消費電力は60W程度(約2倍)である．CPUを2個使っているので60Wとなり，他社並みという意見は置いておく．

そして，Power4が低消費電力だったため，その技術はPowerPCにも流用されるようになる．2001年のMicroprocessor Forumでは，Power4の低電力技術を応用した携帯機器用MPUであるPowerPC405LPが発表された．1.0V駆動，150MHz動作で200MIPS以上，50mWというから，単位電力当たりの性能ではIntelのXScaleを上回る．なお，380MHz動作では1.8V駆動で500MIPS以上，500mWの性能という．

また，同じMicroprocessor Forumで，IBMは将来的に1GHz動作が可能なPowerPC750FX(コードネームSahara)を発表した．PowerPC750CXeの高速版で，最初は750MHzでの出荷となる．これもPower4の技術を流用したものと考えられる．

Power4に関していえば，2002年秋にPower4を基にしてAltiVecと同様なベクタ命令を内蔵した64ビットPowerPCを発表するという噂が2002年夏頃から出ていた．この新PowerPCは，Power4のDNAを引き継ぐチップになるという．その意味するところは高性能であり，2002年末までには2GHzを達成する見込みとか．また，そのベクタ命令はVMXと呼ばれ(Vector/SIMD Multimedia Extensions)，162種をサポートする．しかも，AltiVecとは高い互換性があるらしい．これを以って，この新PowerPCがMacintoshに採用されるのではという憶測もあわせて飛んでいた(事実，そうなった)．

そして2002年10月14日，Microprocessor Forumに先立ち，Power4を基にしたPowerPCであるPowerPC970が正式に発表された．1.8GHzで動作し，32ビットだけでなく64ビットアプリケーションでの実行も可能である．さらにSMPをサポートする．Power4との最大の違いはCPUコアを一つしか内蔵しないこと，CPUコアの動作周波数が1.8GHzに引き上げられたことである(Power4は1.3GHz)．ベクタ命令に関しては，プレスリリースでは，単にSIMD命令をサポートするとなっている．

また2003年6月23日，AppleはPowerMac G5を世界最高速で世界初の64ビットPCとして発表した．プレスリリースによれば，CPU(PowerPC G5)はAppleとIBMの共同開発となっているので，PowerPC970と考えて間違いない．G4のAltiVecを巡って一度は袂を分けたAppleとIBMだが，なかなかクロックの上がらないMotorolaのG4にしびれを切らせたAppleが，IBMとよりを戻したといったところか．

● MotorolaのPowerPC

一方，MotorolaのPowerPC戦略はどうだろう．2000年時点でのロードマップには第4世代のMPC74xx(G4)に続いてMPC85xx(G5)，MPC86xx(G6)が掲載されている．型番が7000番代から8000番代に変わっているのはコアの変更を意味する．つまり，603eからBook Eアーキテクチャを実装するe500に変更される．G5の動作周波数は800MHzから2GHz以上を目標とし，SMP対応を行う．G6に関してはすべてが未定である．PowerPC G5はMacintoshに搭載される予定で，2003年早々に発表されるはずだったが，その名称はIBMに奪われてしまった．現在ではG5という看板は降ろしている．

なお，Book Eとは，MotorolaとIBMが共同で開発したアーキテクチャで，組み込み制御分野においてMotorola

とIBMの製品に互換性をもたせる目的がある．これはPowerPCアーキテクチャの拡張機能である．

2001年のMicroprocessor ForumでMotorolaも1GHz動作のPowerPCであるMPC8540(e500コアの最初の製品)を発表した．e500コア，256KバイトのL2キャッシュ，PCI-Xコントローラ，DDR-SDRAMコントローラ，Ethernetコントローラ，Motorolaの提唱するRapidI/Oなどを内蔵する．性能は2325MIPS，消費電力は6.5Wである．その前哨戦としてか，1月28日，AppleがPowerMac G4に採用したMPC7455(PowerPC7455，コードネームApollo)では1GHz動作を達成した．32Kバイト＋32KバイトのL1キャッシュと256KバイトのL2キャッシュを内蔵し，2MバイトまでのL3キャッシュをサポートする．このほかに，SMP対応やキャッシュロックなど，ネットワーク応用分野への新機能が加わった点を除けば，従来のMPC7450と変わりない．ただ，SOI(Silicon On Insulator)技術で消費電力を抑えている．さらに，MotorolaはMPC7455と同時に800MHz動作で低消費電力を特徴とするMPC7445も発表している．これはMPC7455と同じCPUコアであるが，組み込み制御分野をねらう廉価版であり，L3キャッシュのサポートがない．当初MPC8540で達成する予定だった1GHz動作はすでに達成したことになる．技術の進歩は速い．MPC8540ではバス速度の改善がメインになると思われる．MotorolaはMPC7455で，PowerPC分野でのIBMに対する優位性をアピールしている．しかし，動作周波数に関していえば，IBMに一歩遅れをとっている．

PowerPC G4は低価格版MacintoshのMPUとして現役を維持しているが，その主流はIBMのPowerPC G5に移りつつある．最近のMotorolaは，自社の強みである通信やネットワーク分野にPowerPCを売り込む方針に転換したように見える．

● PA-RISC

PA-RISC(Precision Architecture RISC)とはHP社のEWSであるHP9000シリーズのアーキテクチャであり，EWSの分野ではかなりの実績をもつ．それでいて，ビット操作命令，ビットフィールド命令，独自機能をサポートするSFU(Special Function Unit)を有し，組み込み制御分野にも適している．事実，1991年に発表されたHD69010(PA/10)はEWSだけでなく組み込み分野への応用も見込んでいた．

PA-RISCのアーキテクチャはHPと契約を結んだメーカーにしか公表されていないので詳細は長らく不明だった(現在は，その概要をHPのサイトで知ることができる)．ただ，初期のPA-RISCではメモリアクセスを非常に高速にし，その代わりL2キャッシュを使用しないことを公言していた．メモリアクセスが十分高速なら容量に制限のあるL2キャッシュは意味がないというわけである．この思想が現在も生きているか否かは不明だが，メモリアクセスを高速化したければ，DRAMでなく(高価でも)SRAMを使えという主張は核心を突いているかもしれない．

1998年に発表されたPA-8500では巨大な4ウェイのL1キャッシュ(命令0.5Mバイト，データ1Mバイト)を内蔵している．2001年に発表されたPA-8700に至っては，命令0.75Mバイト，データ1.5Mバイトと超巨大なL1キャッシュを内蔵する．これはチップ面積の3/4以上を占め，MPUではなくSRAMチップと揶揄する声もあるらしい．ただし，あいかわらず，L2キャッシュはサポートしていない．PA-8700は800MHz以上で動作する．

HP自体はEWS用のMPUをIntelのIA-64に移行することを表明しているので，PA-RISCが幻のアーキテクチャのまま終わってしまう可能性は大きい．とはいえ，IA-64の開発遅れに危機感をもっているのはHPも同様らしい．2000年に発表されたロードマップでは，2001年に800MHz動作のPA-8700，2002年に1GHz動作のPA-8800，そして将来的に1.2GHz動作のPA-8900の開発を行うことが明示されている．PA-8900以降は完全にIA-64に移行する予定であるが，これはItanium，Mckinleyに続く第3世代であるDeerfieldやMadisonのあとになっている．

2004年2月9日，HPはPA-8800を搭載したサーバを発表した．PA-8800はPA-8700を1チップに2個搭載するCMPである．当初予定よりは2年の遅れである．同時にDeerfieldを2個搭載したサーバも発表した．これはPA-RISCからItanium2への移行が予想通り進んでいることを示すものである．ただ，業界筋ではPA-8800はキャンセルされるという予想が大勢だったが，それでも製造に漕ぎ着けるとは大口契約など止められない理由があるのかもしれない．しかし，HPがOpteronを採用したり，Yamhillの発表でItanium2自体の存在価値が危うくなってきている状況では，将来どうなるかわからない．とりあえずはPA-8900を開発するのだろうか．

HPは，PA-RISCからIA-64への以降は非常に簡単だと言っている．なぜなら，IA-64の命令セットのほとんどはPA-RISCのものであり，バイナリレベルの互換性があるという．これに加えて，データの互換性(エンディアンが同じということ？)もあることが特筆すべきこととして挙げられている．

● Alpha

AlphaはDECのAlpha AXPというアーキテクチャに準

拠したRISCである．VAXのユーザーをよりハイエンドのEWSへと導くため，1989年の中頃からプロジェクトが始まった．DECにとっては最初のRISCアーキテクチャではなかったが，商業的に初めて成功したのがAlphaである．これは当初から64ビットアーキテクチャを提供し，64ビットのロード/ストア命令を基本として命令セットが構築されている．命令長は32ビット固定で140種の命令がある．最初のAlpha21064は200MHzという，当時としては信じられない高速動作を実現し，最高速のMPUとしてギネスブックに掲載された．

Alphaは21064(200MHz)に始まり，バイトとワードのデータ型をサポートして動作周波数を向上(最大600MHz)させた21164，新たにモーションビデオ命令(MVI)を追加してさらに動作周波数を向上(700MHz以上)させた21264がこれまでに開発された．この間，「ほとんど誰も欲しがらない最先端技術」の典型と揶揄されながらも，25年のライフサイクルを想定していたらしい．1999年には量産チップとして初めて1GHz動作を達成した．そして，今後も21364，21464と続いていく予定だった．当初予定では，21364が2000年，21464が2001年に登場する予定だったが，開発は順調に遅れてしまったようである．

なお，Alpha AXPアーキテクチャはCPUコアによって区別される．それはEV(Electro Vlassic)の名称で呼ばれる．その数値はDEC(後期にはIBM)の半導体製造プロセスを表している．具体的には，21064/150MHzがEV4，21064/200MHzがEV4S(Sはシュリンクを示す)，21064AがEV45，21164がEV5，21164AがEV56，21264/600MHzがEV6，21264A/700MHzがEV67(0.25 μm)，21264B/1GHzがEV68(0.18 μm)，21364/1.2GHzがEV7(0.18 μm)と呼ばれている．

余談ではあるが，AMDがAthlonのバス仕様としてAlpha EV6の仕様を採用してIntelと差別化を図ったのは有名である．

1998年，CompaqはDECを買収してAlphaアーキテクチャを手に入れた．Compaqはx86チップの大口顧客として有名だが，AlphaチップはEWS用である．それまでMIPS社のMPUを使用していたTandem向けEWSを21264で置き換える予定だったようだ．

そして2001年6月25日，CompaqはAlpha事業をIntelに譲り渡すと発表した．IntelはAlphaの知的資産を獲得すると共に，Itaniumに対する直接的な脅威を消し去ることに成功した．Compaqは「AlphaのエッセンスはItaniumの中で生き続ける」としているが，Alphaアーキテクチャの事実上の消滅である．そして，Alphaの技術は6代目(？)ItaiumであるTukwilaに採用されるという．

また，Compaqは自社のサーバをItaniumに置き換える予定であるが，Alphaの最終版としてEV67とEV7(開発コードMarvel)は開発するらしい．どちらも動作周波数は1GHz以上という．EV7のリリースは2003年の予定で，その間にAlphaの開発陣は順次Intelに移籍してItaniumの開発に携わっていくことになる．マルチスレッド対応が予定されていたEV8の開発は中止された．

まとめ

ほかにもルネサステクノロジのSHシリーズやNECのV800シリーズや，さらにもう少しマイナー路線(？)まで広げると，日本の半導体メーカー各社が何がしかのRISC系マイコンをもっているが，誌面が尽きたので残念だがここまでとする．

Intelに端を発するマイクロプロセッサの歴史は，ほかのアーキテクチャとの攻防があったが，結局はIntelの一人勝ちの状況で進化が進んでいる．プログラムがC言語で開発されるのが主流になった現在では，命令セットアーキテクチャへの関心は薄くなっている．

今後はプロセス技術の進化とともにマイクロプロセッサも進化していく．その先駆者の一つは間違いなくIntelであるが，それと双璧をなすIBMにも，RISCの火を消さないで頑張ってもらいところだ．

あとがき――本書が産まれるまで

本書の原稿を書き始めるきっかけは，コンピュータアーキテクチャ，具体的にはMMUやキャッシュの構成などについて知っている人が思ったよりも少ないということを実感したことだった．会社の後輩から「2ウェイセットアソシアティブってなんですか?」とか「ガード，ラウンド，スティッキってなんですか?」と質問されたとき，最近の若い者はそんなことも知らないのかと思ってしまった．たしかに特殊な知識ではあるのだが，MPUを設計しようと思っている人には必須である．しかし，それを学ぶための書物が少ないのも現実である．

コンピュータアーキテクチャの教科書としては，RISCの産みの親であるHennessyとPattersonが著した，いわゆる『ヘネパタ本』が有名である．この本でだいたいの知識は得ることができるが，高価なのが玉にきずである．もっと手軽に読めるものが欲しいと思う．とはいえ，巷にあふれているパソコン雑誌で，マイクロアーキテクチャの詳細について解説してあるものは少ない．あったとしてもIntelやAMDの新製品に関するアーキテクチャの解説なので，非常に偏っている．x86アーキテクチャがすべてではないのだということを知ってもらうために本書を著した．

個人的には，x86アーキテクチャは非常に複雑だと思う．アドレス変換やメモリ保護など，昔ながらのセグメント方式と矛盾がないように拡張しているので，理解が難かしい．コンピュータアーキテクチャの基礎をx86で学ぼうというのは無謀である．やはり，シングルパイプライン構造のRISCから始めるのが王道であろう．

*　　　　　　*

本書は，Interface本誌2003年の10月号と11月号の2号にわたり企画された特集の原稿をベースに，大幅に加筆修正して単行本化したものだ．実を言うとその特集の原稿そのものも，他社の雑誌に執筆し連載した記事をベースとしている．本来なら，最初に雑誌に掲載された後に単行本化される場合は，その雑誌と同じ出版社から出版されるのが普通であるが，いろいろな事情から出版社が違う場合もある．いわゆる“大人の事情”というやつである．

それゆえ，この原稿が本当に日の目を見るのか，本当に単行本化が決まるのか，筆者は神にも祈る気持ちで審判を待った．こうしてこの文章が読者に読まれているということは，待った甲斐があったというところか．

本書には，筆者のコンピュータに対する知識の8割程度をつぎ込んでいる．筆者の知識そのものといってもいい．一冊の本にまとめてみれば，筆者の知識もこの程度のものかと気落ちしてしまうかもしれないが，本書は筆者の備忘録でもある．自己満足といわれれば，そうかもしれない．そんなものでも買ってくれる読者がいれば嬉しいと思う．

参考文献

参考文献を明示した章もあるが，ほかの章でも以下の文献を適宜参考にしている．このほかにも，特に挙げてはないが，インターネット上の各Webサイトで行われているコンピュータ関連の種々の解説には大いにインスパイアされるものがあった．

▶アーキテクチャ全般

1) 馬場敬信；コンピュータアーキテクチャ(改訂2版)，オーム社，2000年.
2) 神保進一；最新マイクロプロセッサテクノロジ 増補改訂版，日経BP社，1999年.
3) マイク・ジョンソン；スーパースカラ・プロセッサ，日経BP社，1994年.
4) ヘネシー&パターソン；コンピュータ・アーキテクチャ設計・実現・評価の定量的アプローチ，日経BP社，1992年.
5) Stephan B. Furber；比較研究RISCアーキテクチャ 基礎から学ぶプロセッサ設計とVLSIチップの実例，日経BP社，1992年.
6) Harold S. Stone；高性能コンピュータアーキテクチャ，丸善，1989年.
7) 日経データプロ マイクロプロセッサ vol.1，日経BP社，1988-1994年.

▶x86系

1) i486 MICROPROCESSOR HARDWARE REFERENCE MANUAL，Intel，240552-001，1990.
2) 蒲地輝尚；はじめて読む486，アスキー，1994.
3) J.H.Crawford and P.P.Gelsinger；80386プログラミング，工学社，1988年.
4) Robert Franklin Krick and et al.；“TRACE BRANCH PREDICTION UNIT”，US Patent 6,014,742.
5) S.Gochman, ; R.Ronen, ; I.Anati, ; A.Berkovits, ; T. Kurts, ; A.Naveh, ; A.Saeed, ; Z.Sperber, ; and R. Valentine, “The IntelR PentiumR M Processor: Micro architecture and Performance.” Intel Technology Journal. `http://developer.intel.com/technology/itj/2003/volume07issue02/` (May 2003).
6) 元麻布春男；IT管理者のためのPCエンサイクロペディア――基礎から学ぶPCアーキテクチャ入門――（第7回～第15回）(`http://www.atmarkit.co.jp/fsys/rensai/indexpage/index.html`).

▶680x0系

1) モトローラ，MC68030ユーザーズ・マニュアル，1990年.
2) モトローラ，M68000マイクロプロセッサ ユーザーズ・マニュアル 第4版，1984年.
3) モトローラ，MC68020ユーザーズ・マニュアル 第2版，1986年.
4) MOTOROLA，M68000 FAMILY REFERENCE，1990.
5) MOTOROLA，PROGRAMMER's REFERENCE MANUAL，1992.
6) MOTOROLA，MC68040 32-BIT MICROPROCEOOSR USER'S MANUAL，1989.

▶Crusoe系

1) Alexander Klaiber, “The Technology Behind Crusoe Processors Low-power ―x86-Compatible Processors Implemented with Code Morphing Software―”, Transmeta Corporation, January 2000.
2) David R. Ditzel, “Transmeta' s Crusoe: Cool Chips for Mobile Computing”, Transmeta Corporation, Hot Chips 2000 プレゼンテーション資料.

3) Robert F. Cmelik and et al., "Combining hardware and software to provide an improved microprocessor", US Patent 6,031,992.

▶IA-68系

1) Harsh Sharangpani, "Intel Itanium Processor Microarchitecture Overview", Microprocessor Forum 1999 プレゼンテーション資料.
2) "Inside the Intel Itanium2 Processor: an Itanium family member for balanced performance over wide range of applications", a Hewlett Packard Technical White Paper, July 2002.

▶PowerPC系

1) "MPC750 RISC Microprocessor User's Manual", Motorola, 1997.
2) 石井孝利,『「パワーPC」巨大連合の逆襲』, 東洋経済新報社, 1995年.
3) IBM RS/6000とPOWERアーキテクチャの10年間(http://www-6.ibm.com/jp/servers/aix/developer/feature/power10/b01.html)

▶Alpha系

1) Shrewsbury, Massachusetts, "Alpha 21264/EV6 Microprocessor Hardware Reference Manual", Compaq Computer Corporation, DS-0027B-TE.

▶SPARC系

1) "The SPARC Architecture Manual Version 8 Revision SAV080SI9308", PARC International Inc.

▶PA-RIC系

1) "PA-RISC 1.1 Architecture and Instruction Set Reference Manual", Hewlett-Packard, February 1994, 09740-90039.
2) "PA-RISC 8x00 Family of Microprocessors with Focus on PA-8700 Technical White Paper", Hewlett-Packard, April 2000.

▶MIPS系

1) Darren Jones, "The MIPS64 5Kc Processor: The First Synthesizable 64- bit Core", MIPS Technologies Inc., Microprocessor Forum 1999 プレゼンテーション資料.
2) Tom R. Halfhill, "Jade Enriches MIPS Embedded Family――First Synthesizable Cores From MIPS Implement New 32-Bit Architecture――", Microprocessor Report Volume 13, Number 07.
3) "MIPS R10000 Microprocessor User's Manual Version 2.0", MIPS Technologies Inc., 1996.
4) 日経データプロ・マイクロプロセサ,『製品概要レポート μPD30400(VR4000PC), μPD30401(VR4000SC)』, MC1-404-301～314, 1993年.
5) 日経データプロ・マイクロプロセサ,『製品概要レポート μPD30410(VR4400PC), μ PD30411(VR4400SC), μPD30412(VR4400MC)』, MC1-404-351～362, 1993年.
6) "An Architecture Extension for Efficient Geometry Processing" (www.mips.comより)

▶SuperH系

1) SuperHプロセッサ, CQ出版社, 1999年.
2) 日立SuperH RISC engine SH-4ハードウェアマニュアル SH7750, ADJ-602-148A, 日立製作所, 1998年.
3) 日経エレクトロニクス, 開発ストーリ「SHマイコン開発」

▶ARM系

1) Steve Furber；ARMプロセッサ, CQ出版社, 1999年.
2) Steve Furber；改訂 ARMプロセッサ, CQ出版社, 2001年.
3) Simon Segars, "ARM1136J-S and ARM1136JF-S First ARMv6 Architecture Cores", October 2002, Microprocessor Forum 2002のプレゼンテーション資料
4) M. Levy, "ARM EMBRACES SIMD SUPPORT", MICROPROCESSOR REPORT, Jan.2, 2001.

▶Athlon系

1) "QuantiSpeed Architecture", AMD, January 30, 2002.
2) "The AMD Athlon XP Processor", AMD, June 10, 2002.
3)『AMD Athlon Perfect Book』, ソフトバンクパブリッシング, 1999年.

▶Hammer系

1) Fred Weber, "AMD's Next Generation Microprocessor Architecture", October 2001, Microprocessor Forum 2001のプレゼンテーション資料.

▶x86-64系

1) "x86-64 Technology White Paper", AMD.
2) "The AMD x86-64 Architecture Programmers Overview", AMD, Publication # 24108 Rev:C, January 2001.

▶Power4系

1) J. M. Tendler and et al., "POWER4 system microarchitecture", IBM Journal of Research and Development, Volume 46, Number 1, 2002.

▶FPU系

1) 数値演算プロセッサ, 別冊インターフェース, CQ出版社, 1987年.

▶高速演算器系

1) Kai Hwang；コンピュータの高速演算方式, 近代科学社, 1980年.

▶VR4131系

1) 正木他；携帯情報端末・組み込み用周辺機能内蔵64ビットMIPSプロセッサ V_R4131, NEC術報Vol.54, No.3, pp.19-24, 2001.
2) 中川靖；64ビットRISCマイクロプロセッサ V_R4131, NEC Device Technology 2001, No.74.

▶V810系

1) 山畑他；2.2V動作を可能にしたV810 個人向け通信機器や家庭用マルチメディア機器などをねらう, 日経エレクトロニクス, No.568, pp.113-122, 1992年.

▶V60/V70系

1) V60, V70ユーザーズ・マニュアル アーキテクチャ編, IEM-949G(第8版), August 1991 P, 日本電気.

▶その他の情報

1) IC Collection(http://www.st.rim.or.jp/~nkomatsu/ICcollection.html)
2) Chip Architect(http://www.chip-architect.com/news/2003_03_06_Looking_at_Intels_Prescott.htm)
3) 旧ZDNN(現TImedia)(http://www.itmedia.co.jp/)
4) PC Watch(http://pc.watch.impress.co.jp/)
4) Hisa Ando氏の個人ページ(http://www.geocities.co.jp/SiliconValley-Cupertino/6209/)

■ 筆者略歴

中森 章(なかもり あきら)

1960年	岡山生まれ
1979年	東京大学理科I類入学 化学実験で使用したプログラミング電卓が初めてのプログラミング経験
1981年	同大学工学部進学．大型計算機に興味を持ち，アセンブラでプログラミング経験を積む．Z80を見たときレジスタの少なさに愕然
1982年頃	C言語を覚え始める．この頃から某パソコン雑誌で雑文を書き始める
1983年	同大学大学院工学系研究課入学．だらだら過ごす
1985年	某電気メーカに就職しマイクロプロセッサ(CISC)の開発に従事．マイクロコードをひたすら書く この頃に68000を知る．アーキテクチャの美しさに感激
1995年頃	世の風潮に迎合し，RISCプロセッサの開発に転向
現在	組み込み向けRISCプロセッサの設計開発に従事

Interface増刊
マイクロプロセッサ・アーキテクチャ入門

2004年4月1日　発行

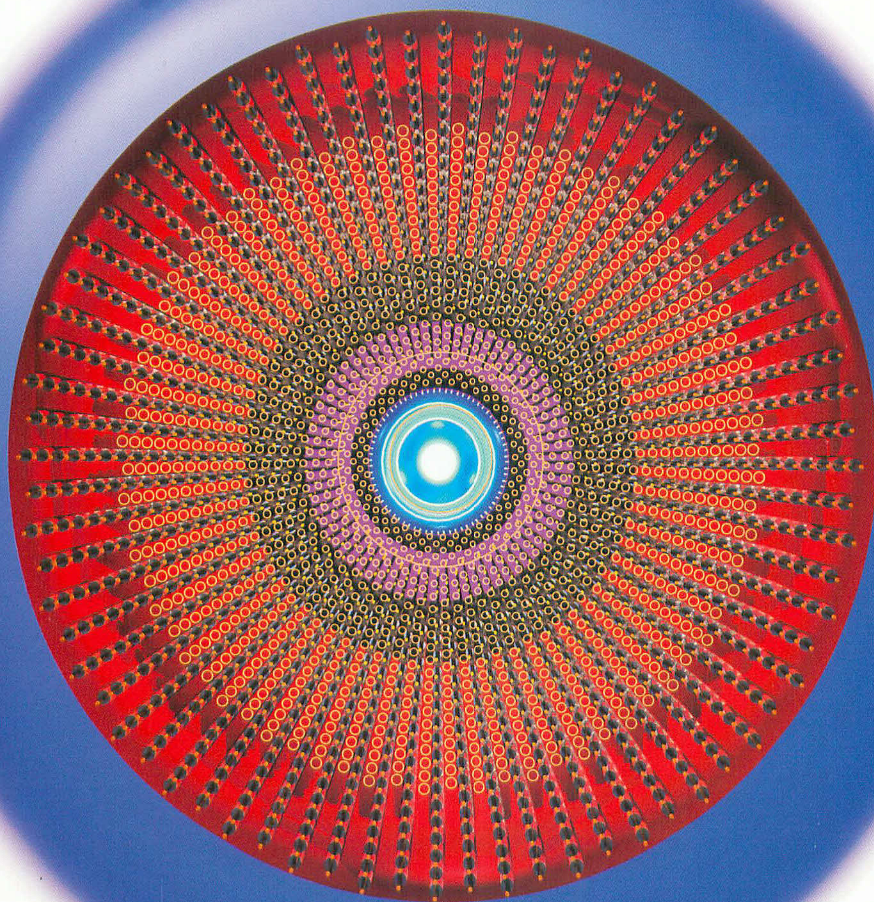
編集人　山　形　孝　雄
発行人　増　田　久　喜
発行所　CQ出版株式会社

〒170-8461　東京都豊島区巣鴨1-14-2
電話　出版部　03-5395-2122
　　　販売部　03-5395-2141
　　　振　替　00100-7-10665

(定価は裏表紙に表示してあります)

乱丁，落丁本はお取り替えします

制作　(有)みやこワードシステム
印刷・製本　大日本印刷(株)
Printed in Japan



インターフェース増刊 テックアイ Vol.20

CQ出版社 〒170-8461 東京都豊島区巣鴨1-14-2 TEL.(03)5395-2141(販売部)

定価2,310円 本体2,200円

雑誌 01620-4

Ⓛ-2004. 5/15

T1101620042315

